Generating Commonsense Explanations with Answer Set Programming

PhD Thesis

UNIVERSIDADE DA CORUÑA

Brais Muñiz Castro

2024

Generating Commonsense Explanations with Answer Set Programming

Brais Muñiz Castro

PhD Thesis / 2024

Advisors:

José Pedro Cabalar Fernández

Gilberto Pérez Vega

PhD in Computer Science

UNIVERSIDADE DA CORUÑA

José Pedro Cabalar Fernández, Chair Professor at the Department of Computer Science and Information of Universidade da Coruña,

and

Gilberto Pérez Vega, Professor at the Department of Computer Science and Information of Universidade da Coruña,

HEREBY CERTIFY

that the present Doctoral Thesis, ***Generating Commonsense Explanations with Answer Set Programming***, submitted to the Universidade da Coruña by Brais Muñiz Castro, has been carried out under our supervision and fulfils all the requirements for the award of the degree of *PhD in Computer Science with International Mention*.

José Pedro Cabalar Fernández	Gilberto Pérez Vega
Advisor	Advisor

To Tesi.
Thank you for bringing happiness to my heart.
I need you. You are my home.

Acknowledgments

Every day, I am acutely aware that the majority of what I have been able to achieve, and will achieve, is thanks to the people around me who support and help me, rather than solely due to my own efforts. Please, let the individuals I will mention next know that my words below come from the depths of my sincerity.

I would like to start by expressing my heartfelt gratitude to my thesis advisors, without whom this work would not have been possible. I feel incredibly fortunate to have had you as my mentors. Gilberto, your support and guidance throughout these years have been invaluable. Especially regarding teaching, where you have taught me countless things and where I consider you my role model. However, above all, I want to sincerely thank you for your unwavering care and concern for me over the years. I want you to know that your selfless affection has been a source of strength that has helped me persevere and reach this point. Pedro, I am also deeply grateful to you. The time and effort you have dedicated to me and this project are immeasurable, and I doubt I will ever be able to fully repay you. Even in times when I may have fallen short or made mistakes, you have always been there to help me without hesitation and with the best attitude. Above all, I greatly appreciate that you have always cared about seeking the best for me.

I also want to extend my gratitude to Feli, Conchi, and Javier, who also cared for me deeply. Sincerely: your gestures of affection have not gone unnoticed, and I appreciate them more than words can express. Thank you all five from the bottom of my heart for taking care of me. I would also like to thank my friends and colleagues at the IRLab: Paloma, Manu, David, Anxo, Alfonso, Jorge, Eliseo, Juan, Antón, Edu, and everyone else. Thank you for making work so enjoyable. On the toughest days, when motivation is lacking, knowing that these incredible people are there waiting for you makes work effortless. Also, I would like to mention the rest of my fellow professors in the algebra department: Ana, Pedro, Óscar, and everyone else. I have had the pleasure of learning from them, and I hope to continue doing so.

I also want to express my gratitude to the people at the Knowledge Processing and Information Systems group at the University of Potsdam, where I carried out my stay. I will never forget how welcomed I felt there.

I want to thank my lifelong friends, whom I love dearly. Having you all by my side is incredibly important to me. Knowing that our friendship will never break gives me so much strength. I will always strive to live up to the love you've shown me.

Finally, I reserve the greatest gratitude for my incredible family. Mom, Dad: know that, to me, you are the epitome of goodness, and I can only hope to one day become even half as amazing as you both are. My dear wife, Tesi, thank you for always being by my side and loving me so much. I can rest assured knowing that, even if everything else were to fail, I would still have you.

Abstract

In this thesis, we explore the notion of commonsense explanation in the context of Artificial Intelligence by extending the formalism of Answer Set Programming (ASP) with formal annotations. To this aim, we define the concept of support graphs to account for the multiple explanations for each model of a logic program, and we provide different operations to filter irrelevant information from the graphs. These definitions are implemented in a tool called `xclingo` that additionally allows the specification of natural language, commonsense explanations. `xclingo` obtains the support graphs via an ASP meta-encoding that is proved to be correct. We study different examples in the context of ASP such as planning, problem-solving, or diagnosis, among others, and we analyze the effect of alternative annotations for the same scenario, illustrating the need for explanation design. Additionally, we address the generation of non-technical explanations of Machine Learning models for real users in a pair of problems from other disciplines (Medicine and Pharmacy), covering both symbolic and sub-symbolic learning algorithms.

Resumen

En esta tesis, exploramos la noción de "commonsense explanation" en el contexto de la Inteligencia Artificial mediante la extensión del formalismo Answer Set Programming (ASP) con anotaciones formales. Con este objetivo, definimos el concepto de "support graphs" para obtener múltiples explicaciones de cada modelo de un programa lógico, y proporcionamos diferentes operaciones para filtrar la información irrelevante de estos grafos. Dichas definiciones son implementadas por una herramienta llamada `xclingo` que adicionalmente permite la especificación de explicaciones "commonsense" en lenguaje natural. `xclingo` obtiene los "support graphs" empleando un meta-programa ASP, cuya corrección es demostrada. Estudiamos los diferentes ejemplos en el contexto de ASP tales como planificación, resolución de problemas o diagnóstico, entre otros, y analizamos el efecto de diferentes anotaciones para el mismo escenario, ilustrando la necesidad de diseñar las explicaciones. Adicionalmente, abordamos la generación de explicaciones no técnicas de modelos de Aprendizaje Automático con usuarios reales en dos problemas de otras disciplinas (Medicina y Farmacia), cubriendo tanto algoritmos de aprendizaje simbólico como subsimbólico.

Resumo

Nesta tese, exploramos a noción de “commonsense explanation” no contexto da Intelixencia Artificial mediante a extensión do formalismo Answer Set Programming (ASP) con anotacións formais. Con este obxectivo, definimos o concepto de “support graphs” para obter múltiples explicacións de cada modelo dun programa lóxico, e proporcionamos diferentes operacións para filtrar a información irrelevante de destes grafos. Ditas definicións son implementadas por unha ferramenta chamada `xclingo` que adicionalmente permite a especificación de explicacións “commonsense” en linguaxe natural. `xclingo` obtén Os “support graphs” empregando un meta-programa ASP, cuxa corrección é demostrada. Estudamos os diferentes exemplos no contexto de ASP tales como planificación, resolución de problemas ou diagnóstico, entre outros, e analizamos o efecto de diferentes anotacións para o mesmo escenario, ilustrando a necesidade de deseñar as explicacións. Adicionalmente, abordamos a xeración de explicacións non técnicas de modelos de Aprendizaxe Automático con usuarios reais en dous problemas de outras disciplinas (Medicina e Farmacia), cubrindo tanto algoritmos de aprendizaxe simbólico como subsimbólico

Contents

List of Figures

List of Programs

Chapter 1

Introduction

Within the last few years, we have witnessed how Artificial Intelligence (AI) has mastered tasks considered unthinkable to be performed by anything different from a trained human. A good example of this is Generative AI [23], which is able to write and comprehend natural language text to, clone voices, as well as to generate all kinds of graphics such as drawings, photorealistic pictures, or even 3D models. The advancement of AI has been so rapid that research struggles to identify potential positive applications that may arise. These include fraud or cyberbullying detection on social networks, personalized education, and simultaneous translation, to name three examples. However, like a double-edged sword, it is equally true that these technologies can also be maliciously exploited in many ways such as political manipulation, automated cybercrime, or identity theft.

Concerning that, countries are increasing their efforts to create regulations [28]. During December 2023, for instance, the Council and the Parliament of the European Union (EU) approved the first comprehensible law on AI [27] (the so-called *EU AI Art*). In this document, AI systems are classified into different risk levels, with their corresponding regulatory frameworks to control or mitigate them. Among the criteria used to identify the level of risk of a particular system are those data-related (such as privacy and the absence of unfair biases) but also, and perhaps more importantly, the domain or field in which these systems are used also comes into play, with some being identified as higher-risk areas. These include the legal, administrative, educational, military and/or police, and healthcare domains. Associated with those risks, several requirements that the systems must comply with are defined. In particular, according to points 1 and 2 of Article 13 of the EU Parliament's proposal, transparency and comprehensibility of the systems are required.

> *Article 13.1. High-risk AI systems shall be designed and developed in such a way as to ensure that their operation is sufficiently transparent to enable users to interpret the system's output and use it appropriately. ...*
>
> *Article 13.2. High-risk AI systems shall be accompanied by instructions for use in an appropriate digital format or otherwise that include concise, complete, correct and clear information that is relevant, accessible and comprehensible to users.*

European Commission proposal on regulating AI Systems [27].
Proposed in 2021, approved in December 2023.

This points in the direction of the *eXplainable Artificial Intelligence* (XAI) research field, which has experienced an unprecedented increase in research interest in recent years. On the one hand, efforts from computer science are made to achieve transparent and comprehensible AI algorithms. On the other hand, and perhaps more importantly, actors from the social sciences such as philosophers or sociologists try to define such concepts in the first place from an ethical and humanistic point of view. As an example, the report by V.Dignum [34] names three main requirements regarding the interpretability of AI systems:

1. Transparency

   > *Indicates the capability to describe, inspect and reproduce the mechanisms through which AI systems make decisions ...*

2. Responsibility

   > *Refers to the role of people themselves in their relation to AI systems. ...*
   > *Responsibility in AI is also an issue of regulation and legislation, in particular where it respects liability ...*

3. *Accountability*

   > *Accountability is the capability to give account, i.e. to be able to report and explain one's actions and decisions. To ensure accountability decisions should be derivable from, and explained by, the decision-making mechanisms used. ...*
   > *In developing explanation mechanisms, it is important to be mindful that the explanations should be comprehensible and useful to a human, ...*
   > *Explanation is relevant for trusting AI systems for a number of reasons. Firstly, explanations can reduce the opaqueness of a system, and support understanding of its behavior and its limitations. ...*

From the definition, the term *Responsibility* seems to fall more on the side of ethics, regulations and legal fields, aiming to control how the AI systems are used, with which purposes, by whom and finally, who and how are they affected. *Transparency* and *Accountability*, however, are challenges that can be tackled more from a more technical side, by providing AI systems and algorithms with such capabilities. We would refer to a system as *transparent* when the mechanism [1] that relates its input into its output is documented and clear enough so that each step is fully known and reproducible. In contrast, an accountable system possesses the ability to *self-explain* in a way that may not only describe its technical behavior but can also justify its decisions or results in a comprehensible and useful way for human actors to trust the system.

Unfortunately, as Tim Miller claims in his influential survey on XAI [75], *"it is fair to say that most work in explainable artificial intelligence uses only the researchers' intuition of what constitutes a 'good' explanation"*, instead of building on the vast existing research on the topic from the social sciences. To gain user's trust in the systems is essential that we look at how real people define, generate, select, evaluate, and present explanations, rather than defining arbitrary *ad-hoc* approaches. As a conclusion, [75]

[1] This includes a procedure, an algorithm, or perhaps an inference method

highlights the following properties of explanations that XAI systems should pursue to enhance user trust.

> (1) *Explanations are contrastive:* the research suggests that people tend to ask for explanations in response to abnormal or unexpected observations with respect to their own beliefs.

In other words, people do not simply ask why some event P occurs. In contrast, they typically ask "*Why P*" in contexts where they expected an event Q to occur, but are instead confronted with a situation where an unexpected event P does hold and Q does not. Therefore, the real question they need an answer to is "*Why P instead of Q?*". Note that this is different than simply asking *Why not Q?*, because it implies that the asking user considers a default, expected world that has been broken. Indeed, it is from the differences between the actual world (where P holds) and the default, expected world (where Q holds) that people start to build explanations. Take for instance the following example from [75].

> *... why the Challenger shuttle exploded in 1986 (rather than not exploding, or perhaps why most other shuttles do not explode). The explanation that it exploded "because of faulty seals" seems like a better explanation than "there was oxygen in the atmosphere".*

Explanation in Artificial Intelligence: Insights from the social sciences
Tim Miller. Published in October 2018.

In the example, P is the explosion of the Challenger shuttle, which happened in the actual world, whereas Q could be that the shuttle do not explode which was something expected by the user. *"There was oxygen in the atmosphere"* is not considered a good explanation because both the actual world and the default world assumed by the asking actor share this. Conversely, the fact that seals were faulty is considered *abnormal* with respect to the actor's assumptions and therefore is perceived by her as a satisfying explanation.

Of course, strictly speaking, the oxygen in the atmosphere is a necessary cause for the explosion to occur, as other (perhaps impossible to count) non-mentioned causes are. In fact, handling the complete chain of causes quickly becomes impossible for humans in real scenarios. This becomes worse if we also try to take into account all the possible causal inhibitors that could prevent the explosion from happening, including those the less related to the actual world. For instance, arguing the fact that it wasn't stormy the day the shuttle launched as a reason for the explosion to happen because it wouldn't have taken off in that case. This connects with the other two properties identified by [75].

> (2) *Explanations are selected (in a biased manner)*: explanations rarely consist of the actual and complete causes of an event, rather than a (biased) selection of a subset of causes based on the context and several other criteria such as temporal proximity, necessity, sufficiency, abnormality, etc.

> (3) *Explanations are social*: when a system explains a result, a transfer of knowledge happens from the system to the user as part of a (people-like) interaction. As such, the information presented is relative to the system's beliefs about the explainer's beliefs.

In other words, explanations are both context-dependent and user-dependent. When people try to find the reasons for an unexpected event, they first try to imagine *close* contrastive scenarios that successfully explain it. This relates to *abductive reasoning* and/or *causal simulation*. However, in this process, the exploration of the hypothetical worlds is not arbitrary but carefully guided by the person's beliefs on the relevance of the different possible causes, which change depending on the context. In other words, the imagined contrastive scenarios are not only *close* but *relevantly close*. This selection of relevant causes not only influences the finding of valid contrastive worlds but also the information ultimately included in the explanation, often also tailored to the receiver's knowledge and perspective. The following quote from [62] (also retrieved from [75]) greatfully illustrates this.

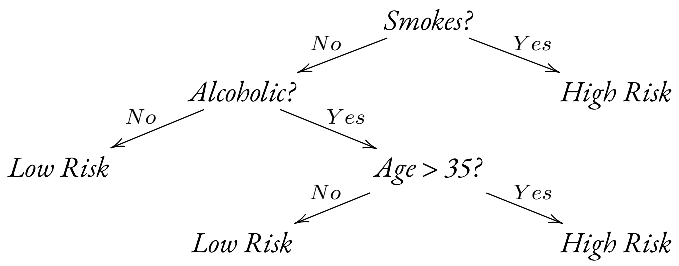
> *There are as many causes of x as there are explanations of x. Consider how the cause of death might have been set out by the physician as 'multiple haemorrhage', by the barrister as 'negligence on the part of the driver', by the carriage-builder as 'a defect in the brakelock construction', by a civic planner as 'the presence of tall shrubbery at that turning'. None is more true than any of the others, but the particular context of the question makes some explanations more relevant than others.*

Patterns of Dicovery
Norwood Russell Hanson. Published in 1958.

In [75] several criteria such as temporal proximity, necessity or sufficiency (among others) are identified as often good filters for relevant causes, whereas others like probability are shown as less useful. In fact, in his final major finding, Miller contrasts probabilities with cause-effect when evaluating a good explanation.

> (4) *Explanations are causal*: while truth and likelihood are important in explanation and probabilities really do matter, referring to probabilities or statistical relationships in explanation is not as effective as referring to causes. The most likely explanation is not always the best explanation for a person, and importantly, using statistical generalizations to explain why events occur is unsatisfying unless accompanied by an underlying causal explanation for the generalization itself.

Unfortunately, Machine Learning (ML) algorithms cannot extract any causal relations from data (since they cannot perform interventions [80, 82]), and so, explanatory approaches for those AI models do not rely on any causal knowledge either. For instance, the SHAP [68] technique assigns a value to each input variable of an instance that represents the average contribution of that variable to the final decision, but this is based on correlation rather than any learned causal knowledge. Besides, another well-known XAI technique LIME [88] explains complex non-interpretable AI models by computing simpler transparent models trained on the predictions of the former. However, even for transparent simpler AI algorithms like Decision Tree (DT), explanations extracted from them are not always causally correct. Take for instance the following DT.



The tree identifies different risk classes for an arbitrary patient based on whether he smokes, whether she is an alcoholic and her age. From such a tree, one can conclude that *Do not smoking* is a reason to have *High Risk* (by sequentially answering No-Yes-Yes to the conditions in the tree) or that *Being Alcoholic* is a reason to have *Low risk* (by answering No-Yes-No). These conditions learned by the algorithm come from statistical relations in the training data that help the DT to better differentiate the instances of each class. However, by relying only on this information and not taking causal relations into account, one may reach absurd conclusions. To quote Judea Pearl in his celebrated book "The Book of Why" [82].

A world governed solely by probabilities and correlations would be a strange one indeed. For example, patients would avoid going to the doctor to reduce the probability of being seriously ill; cities would dismiss their firefighters to reduce the incidence of fires; doctors would recommend a drug to male and female patients but not to patients with undisclosed gender; and so on.

The Book of Why
Judea Pearl and Dana Mackenzie. Published in June 2018.

To summarize, the problem of users' lack of trust in AI seems to lie in the fact that while AI speaks in the language of correlations, statistics and probability, people only understand the language of causality. Controversely, relying only on human-generated natural language to teach AI systems to explain themselves does not seem a good option either. This is the case of *Large Language Models* (LLM) systems [25] like GPT and its widely known chat application ChatGPT. During his talk in Vienna regarding XAI in May 2022, Edward E. Lee made a prediction [65] suggesting that the insistence on explanations from experts may lead to the training of generative AI systems to produce any convincingly plausible content. This prediction materialized later in the same year, 2022, with the public release of ChatGPT [2]. This development enables anyone to generate a paragraph justifying why something should be done with the same ease as producing another justification for doing the opposite (this capability has already been exploited, for example, in the generation of legal claims). Many of these generated contents rely on fallacious reasoning, erroneous computations, or nonexistent facts (referred to as *hallucinations*). However, discerning their fraudulent components is not always straightforward for humans at first glance.

[2] `https://chat.openai.com`

Faced with the inability to generate explanations that generate confidence in users, some authors, as for instance Ghassemi et al. [58], are beginning to believe that the solution is not only to offer convincing explanations but to design systems that are auditable and verifiable both internally and externally. It, therefore, seems clear that it is important for AI systems to have some kind of causal knowledge to support their decisions, either learned autonomously from training or managed by experts. Even more so in the case where the generation of explanations is an essential requirement.

1.1 In the Quest for Commonsense Explanations

The discipline of AI that is very suitable to support this causal knowledge is *Knowledge Representation and Reasoning* (KRR), a field that has been central to AI since its inception by John McCarthy [73]. KRR involves performing inference on explicit representations of knowledge using computational logic. Several specialized disciplines which have matured significantly over time such as Non-monotonic Reasoning, Ontologies, Uncertainty, and Causality, take part in the KRR field. McCarthy introduced the term *'Commonsense Reasoning'* at the birth of the KRR field, emphasizing the need to formally represent human-like thinking in machines. By this term, McCarthy emphasizes that, to achieve human-like thinking machines, we must formally capture human-like reasoning. This is similar to what, as we have already explained, Miller recently tried to establish in his survey [75]: that we need to look at how people explain to teach machines how to explain.

Definition 1 (Commonsense Explanation)**.** *As a parallelism, we propose the notion of 'Commonsense Explanation' as an explanation from a system to a human, that provides accountability to the system. A commonsense explanation is based (fundamentally) on actual causal knowledge of the real-world domain in human understanding terms. Following Miller's findings (2) and (3) a commonsense explanation must be context and user-dependent, carefully selecting the causal knowledge used in it as well as the language and tone with respect to the system's purposes and the system's beliefs about the target human actor.*

A simple idea that perhaps captures most of what we try to establish with this definition is the idea of a *explanation Turing test*. That is: if a user cannot say whether the explanation was generated by a human or a system, then it is a commonsense explanation. Let this intuition be the ultimate goal of our approach.

Indeed the main goal of this dissertation is to develop this notion of explanation under the KRR paradigm. This will be done from a fundamentally practical perspective, by the implementation of tools for computing such explanations and their practical application to different AI problems and systems. The KRR discipline we have chosen to work with the explanations is *Non-Monotic Reasoning* (NMR) [74]. This area of study emerges within artificial intelligence in the late 1970s and early 1980s driven by the efforts to model human-like reasoning. In contrast with classical logic, where deductions are always true even in the presence of additional information, Non-monotonic reasoning deals with reasoning under uncertainty and incomplete information. This seems to be suitable for commonsense reasoning which involves reasoning about everyday situations and making judgments based on experience. This approach is used nowadays for all kinds of purposes and applications.

Answer Set Programming (ASP) stands out as one of the most successful paradigms for practical KRR and declarative problem solving, grounded in NMR [18, 54]. ASP offers a flexible KRR framework built on a solid mathematical foundation, leveraging stable models semantics [57] for logic programming and fully characterized by Equilibrium Logic [79]. In ASP, knowledge is represented as a logic program, that is, a set of rules of the form:

$$H \leftarrow B$$

where H (namely the *head* of the rule) is a set of atoms and B (namely the *body* of the rule) is a set of literals such that we can some atom in H is true if all literals in B hold. For instance, the rule

$$\texttt{is\_on(bulb)} \leftarrow \texttt{closed(switch), not broken(relay).} \qquad (1)$$

can be interpreted in natural language as *"the light bulb is on if the switch is closed and the relay is not broken"*. From a rule with an empty body B (called a *fact*), its head H is considered true as in the following example.

$$\texttt{closed(switch)} \leftarrow \qquad (2)$$

Rules (1) and (2) together form an ASP program that, after solving produces the unique solution (i.e. Answer Set) A_1 as a result.

$$A_1 = \{\text{closed(switch)}, \text{is\_on(bulb)}\}$$

Note that an ASP program can have from none to several answer sets. However, in the additional presence of the fact `broken(relay)`, we would have obtained the following answer set instead.

$$A_2 = \{\text{closed(switch)}, \text{broken(relay)}\}$$

Due to the accessibility and the efficiency of ASP solvers such as clingo [51, 55], ASP-based tools are widely applicable across diverse domains [44], including multi-robot pathfinding, product configuration, spacecraft system diagnosis, phone call routing, transportation line scheduling, biochemical reactions modeling, and workforce management. Given this wide use, it is natural that, in the context of the rise of the XAI, the use of ASP for the generation of explanations has attracted much interest in the past decade. In particular, Jorge Fandinno's PhD [47], which constitutes the theoretical precedent for the work undertaken within this dissertation, interprets ASP programs as cause-effect relations and develops causal semantics based on this. Indeed, the use of ASP rules to represent causal relations is straightforward, just by rethinking the meaning of the rule as B *causes* H. For instance, rule (1) could be rethought as *"closing the switch causes the light bulb to be on when the relay is not broken"*.

Moreover, in the recent survey [48] (published in 2019), up to six different approaches to the explainability of ASP programs are considered. Since that time, new approaches like `s(CASP)` [7] or `xASP` [4] (both will be visited later in the Chapter 6) have emerged, while other already existing have evolved. In particular, `xclingo` [21] is one of these tools, that implements the approach proposed in this dissertation.

1.2 Motivation

We have introduced the notion of *Commonsense Explanation* in Definition 1, inspired by what literature says to characterize a trustable XAI system, including the European new Law on AI requirements and most important properties on explanations given by literature on social sciences, such as the notions of *transparency* and *accountability*. To that aim, we have seen how technological support is necessary to represent and introduce causal knowledge into the systems, and how the ASP paradigm constitutes a good candidate for it. The core of this dissertation will be focused on developing the practical tools to obtain commonsense explanations from (and for) ASP programs.

Take the following example Example 1 from [11], where an agent operating a circuit has to diagnose why a bulb is not lit as expected.

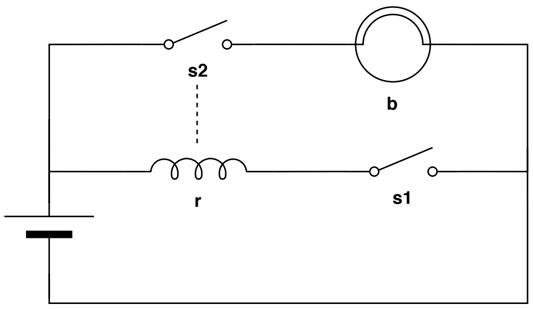


Figure 1.1: A circuit with a bulb b, a relay r and two switches, $s1$ and $s2$.

Example 1. (From [11]) *Consider a system S consisting of an agent operating an analog circuit from Fig. 1.1. We assume that switches s_1 and s_2 are mechanical components that cannot become damaged. Relay r is a magnetic coil. If not damaged, it is activated when s_1 is closed, causing s_2 to close. Undamaged bulb b emits light if s_2 is closed. For simplicity of presentation, we consider the agent capable of performing only one action, $close(s_1)$. The environment can be represented by two damaging exogenous actions: brk, which causes b to become faulty, and srg (power surge), which damages r and also b assuming that b is not protected. Suppose that the agent operating this device is given the goal of lighting the bulb. He realizes that this can be achieved by closing the first switch, performing the operation, and discovering that the bulb is not lit.* □

We will use this example to illustrate to contrast the notions of solution, technical explanation and commonsense explanation. Say that, after some KR effort, a KR engineer writes a program P that captures the diagnosis reasoning needed to solve the example's problem. We do not show the particular implementation of program P as we want to focus on the intuitive comprehensibility of different possible outputs. After solving P with an ASP solver (we will use `clingo`[52], in particular), we obtain the answer sets shown in Output 1.1. In it, we found that there are 3 valid diagnoses: `Answer`

```
Answer: 1
o(toggle(s1),1) o(break,1) o(surge,1) h(relayline,off,1) h(light,off,1)
Answer: 2
o(toggle(s1),1) o(surge,1) (relayline,off,1) h(light,off,1)
Answer: 3
o(toggle(s1),1) o(break,1) h(relayline,on,1) h(light,off,1)
SATISFIABLE
```

Output 1.1: Answer sets for program P, solving Example 1 obtained using `clingo`.

`1`, where the bulb was broken and there was a power surge, meaning that both the bulb and the relay are off; `Answer 2`, where only the power surge happened, meaning that both the bulb and the relay are off; and finally, `Answer 3`, where only the bulb was broken, meaning that both the bulb and but relay are on.

Unfortunately one cannot unravel the reasons why the light is off by just looking at the answer sets answers depicted in Output 1.1. What both approaches [47], and the one taken in this dissertation, do is to build graphs representing the causal chains that explain the different events computed by P. We define our notion of such graphs and refer to it as *Support Graphs* (Definition 6 in Section 3.2) where each node represents a true atom in an answer set, and the edges are drawn from the atoms in the body of a rule to the atoms in its head. However, the way to compute such graphs is not trivial. Indeed, several other approaches apart from [47] and our approach use similar graph-like explanations, also obtained from the program and a particular answer set and they do not coincide with our notion of *Support Graph*. For generating the graphs, a preprocessing of the program is needed to collect the causal relations from the rules. More importantly, how to collect these relations from some particular ASP syntactic constructs such as *choice rules* or *aggregates* is not clear, currently being an open problem in the field for which we propose an approach later in Section 4.3.2.

Even though we consider *Suport Graphs* as explanations for a whole answer set, these are far from being *Commonsense Explanations* since do not meet almost any requirement mentioned in Definition 1. Take for instance the support graph depicted in Figure 1.2, concerning Answer 3 in Output 1.1 with respect to Program P.

Only users knowledgeable in both ASP (and possibly in the particular program) would such an explanation as intuitive or comprehensible. However, they do provide some degree of transparency to an ASP system, as they contain the full derivation trace for every atom in that answer representing how the new atoms are inferred from the knowledge in the program. In fact, by reverse traversing the graph from a given node, one can obtain tree-like explanations of the reasons for a particular atom. Figure 1.3 shows an example of one of these explanations. This kind of explanation, which we refer to as *Technical Explanation* (Definition 2, given below), although is also far from what we could consider a *Commonsense Explanation*, is still important for the internal verification of the system.

Definition 2 (Technical Explanation)**.** *A 'Technical Explanation' fully depicts the underlying mechanisms used by a system to eventually reach a result or a decision. Its purpose is the internal verification of the functioning of the system, and thus it is presented in full detail and in technical language. It is aimed at actors with experience in the underlying technology and/or in the system in particular.*

A good example to illustrate the difference between a technical and a commonsense explanation would be ChatGPT-like explanations. These explanations could be considered commonsense explanations as they are in human terms and are even able to convince real humans with false arguments. The main problem is that, although convincing, they do not offer any warranty of trustability. In contrast, they are not technical explanations at all, since the explanation given has nothing to do with the inner workings of the system that led to their achievement. We aim to equip systems with trustable commonsense explanations providing accountability as well as with technical explanations proving transparency.

With the developed tool `xclingo` we start with support graphs as a basis, we define operations to select the relevant causal knowledge in the graph with respect to the targeted actor's beliefs (fulfilling),

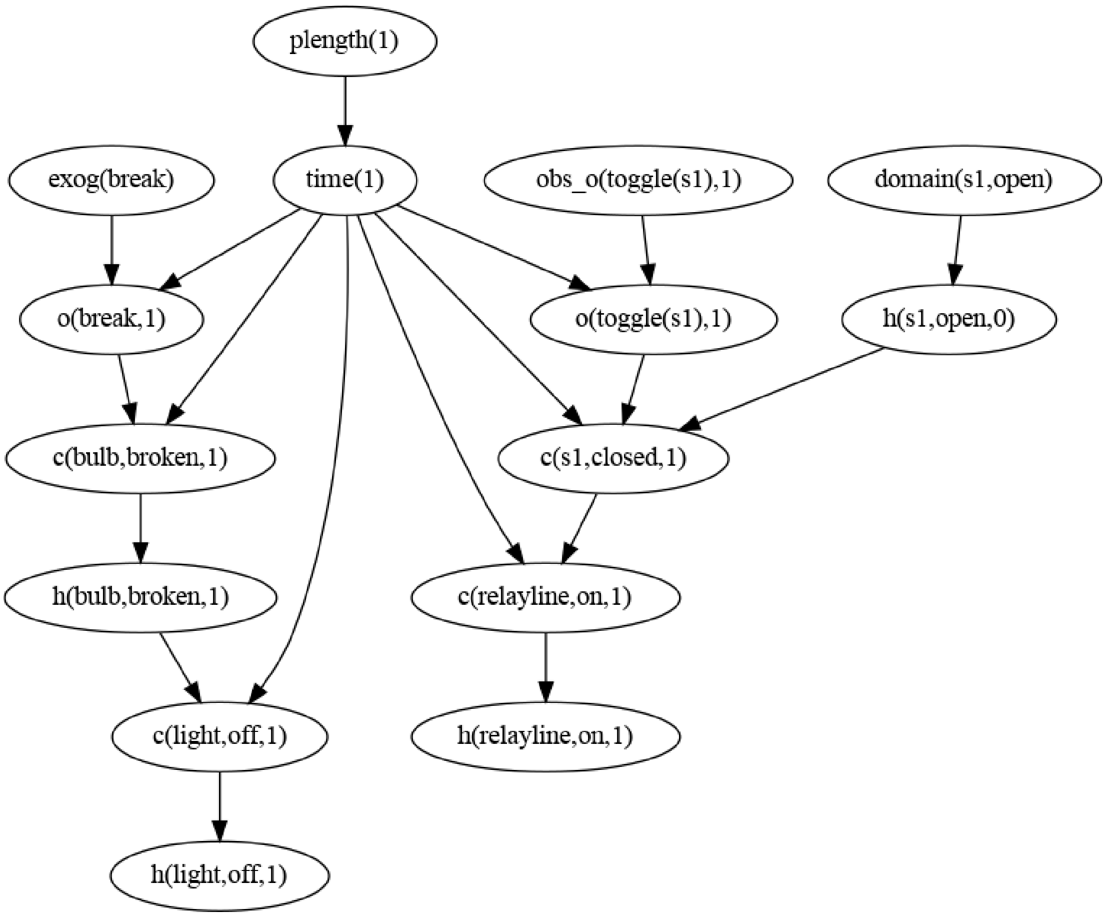


Figure 1.2: *Support Graph* computed by `xclingo` for `Answer 3` in Output 1.1 with respect to Program P.

```
*
|__h(light,off,1)
|  |__c(light,off,1)
|  |  |__h(bulb,broken,1)
|  |  |  |__c(bulb,broken,1)
|  |  |  |  |__o(break,1)
|  |  |  |  |  |__time(1)
|  |  |  |  |  |  |__plength(1)
|  |  |  |  |  |__exog(break)
|  |  |  |  |__time(1)
|  |  |  |  |  |__plength(1)
|  |  |__time(1)
|  |  |  |__plength(1)z
```

Figure 1.3: *Technical Explanation* obtained by `xclingo` for the atom `h(light,off,1)` representing the light beinf off in `Answer 3` of Output 1.1 with respect to program P

```
*
|__"The light is off at 1"
|  |__"The bulb has been damaged at 1"
|  |  |__"Hypothesis: something has broken the bulb at 1"
```

Figure 1.4: A filtered, natural language explanation obtained by `xclingo` from *Support Graph* G_3 in Figure 1.2.

and provide the mechanisms needed to express the explanations in natural language aiming to achieve commonsense explanations. Figure 1.4 shows an example of the explanations that can be achieved with `xclingo`.

Even with the provided tools for selecting and manipulating explanations, we have found that it is not trivial to obtain explanations that fully comply with the final user's expectations. In fact, we claim that right after the effort invested in representing a problem into a program, an additional effort will often have to be made to accommodate the ASP encoding for obtaining the desired explanations. We refer to this new development process as *Explanation Design*, this is a refactorization of original (already correct) encoding, not for maintenance or efficiency purposes, but for ensuring the explanations obtained from the program are commonsense explanations. In Chapter 5 we introduce and detail this process, using a running example to illustrate the different challenges it poses.

Finally, to fulfill the fourth property postulated by Miller in [75] (i.e. explanations are a social interaction), in Section 5.5 we propose a question-answering design for an ASP XAI system. On it, the explanatory process is presented as an interaction between the user and the system, where the system reacts to human demands, sequentially changing the reasoning mode depending on the type of explanatory query the user poses. We propose different types of explanation answers and means to implement them.

1.3 Goals and Structure

This dissertation is posed from a fundamentally practical perspective and it is explicitly divided into two parts. Part I, defines the notion of explanation in ASP (support graphs), describes how `xclingo` implements it, and finally, it provides valuable insights on how to use the tool to obtain commonsense explanations through explanation design for ASP applications. Part II, on the other hand, aims to demonstrate how to obtain explanations in the context of ML applications. In particular, we show two real-world applications, namely: explanations for tree-based ML algorithms applied to liver transplantations and explanations for model-agnostic ML applied to 3D-printing of medicines. To sum up, the goals of this dissertation are:

1. To expand the formalism of logic programming to incorporate new language extensions for designing commonsense explanations based on formal semantics.

2. To implement a tool and to explore its applicability to real problems of logic programming, such as planning, problem-solving, or diagnosis, among others.

3. The study of a methodology that involves the design of understandable explanations for non-technical users as a fundamental part of it.

4. The application of explanation techniques to ML models in real domain with users from outside disciplines.

In particular, this dissertation has produced the following publications within the context of Part I (Explanations for Answer Set Programming):

Cabalar P, Muñiz B. Model Explanation via Support Graphs. Theory and Practice of Logic Programming. 2023 Oct 2:1-4.

Cabalar P, Muñiz B. Commonsense Explanations for the Blocks World. Challenges and Adequacy Conditions for Logics in the New Age of Artificial Intelligence 2023 (ACLAI 2023). Workshop.

Cabalar P, Muñiz B. Explanation graphs for stable models of labelled logic programs. InProceedings of the International Conference on Logic Programming 2023 Workshops co-located with the 39th International Conference on Logic Programming (ICLP 2023), London, United Kingdom, July 9th and 10th 2023 Jul 9.

P. Cabalar, J. Fandinno and B. Muñiz, "A System for Explainable Answer Set Programming", in Proc. of the 36th International Conference on Logic Programming (ICLP'20), EPTCS 325, pp 124-136, 2020.

In addition, it has produced the following publications within the context of Part II[3][4]. (Applications to Explainable Machine Learning).

Ong JJ, Castro BM, Gaisford S, Cabalar P, Basit AW, Pérez G, Goyanes A. Accelerating 3D printing of pharmaceutical products using machine learning. International Journal of Pharmaceutics: X. 2022 Dec 1;4:100120 (Machine Learning pipeline design and experiments)

Cabalar P, Muñiz B, Pérez G, Suárez F. Explainable Machine Learning for liver transplantation. arXiv preprint arXiv:2109.13893. 2021 Sep 28. (unpublished draft)

Castro BM, Elbadawi M, Ong JJ, Pollard T, Song Z, Gaisford S, Pérez G, Basit AW, Cabalar P, Goyanes A. Machine learning predicts 3D printing performance of over 900 drug delivery systems. Journal of Controlled Release. 2021 Sep 10;337:530-45. (Machine Learning pipeline design and experiments; Data preparation)

Elbadawi M, Castro BM, Gavins FK, Ong JJ, Gaisford S, Pérez G, Basit AW, Cabalar P, Goyanes A. M3DISEEN: A novel machine learning approach for predicting the 3D printability of medicines. International Journal of Pharmaceutics. 2020 Nov 30;590:119837 (Machine Learning pipeline design and experiments; Data preparation)

Cabalar P, Martín R, Muñiz B, Pérez G. aspBEEF: Explaining Predictions Through Optimal Clustering. InProceedings 2020 Aug 28 (Vol. 54, No. 1, p. 51). MDPI.

[3]For some publications, the particular contribution of the author is detailed at the end of the citation

[4]In particular for pharmaceutics publications, the order of the authors is established by the amount of work provided by each author

> F.Aguado, P. Cabalar, J. Fandinno, B. Muñiz, G. Pérez and F. Suárez, "A Rule-Based System for Explainable Donor-Patient Matching in Liver Transplantation", in 15th International Conference on Logic Programming (ICLP'19), technical communications applications track, September 20-25, 2019. Electronic Proceedings of Theoretical Computer Science 306, pp. 266-272, 2019.

The rest of this dissertation is organized as follows. Chapter 2 provides some background results that are required in the rest of the dissertation.

Part I starts by providing the definitions of important notions such as support graphs or justified models in Chapter 3. After that, the ASP explainability tool `xclingo` is presented, including the usage of the language extensions as well its implementation based on ASP. Then Chapter 5 provides discusses important topics related to commonsense explanations as well as proposes a methodology to answer real causal questions of different types. Finally, Chapter 6 compares different *state-of-the-art* approaches to ASP explainability both in technical terms and (when possible) in terms of how suitable they are for obtaining *Commonsense Explanations*.

Part II covers the work related to the application explanation techniques to ML applications for obtaining explanations of already trained ML models. Two real proof of concept cases are shown in which we face the task of providing explanations to non-technical users outside the KRR field. Thus, the notions of commonsense explanations become of much greater importance. In particular, Chapter 7 addresses the problem of obtaining comprehensible explanations for Decision Trees in the context of a Decision Support System for Liver Transplantation. On the other hand, Chapter 8 demonstrates the use of a symbolic, model-agnostic explainability approach to explaining ML model predictions for the process of 3D-printing of medicines.

Part III concludes this dissertation by summarizing the contributions and outlining the future research lines.

Chapter 2

Background

2.1 Answer Set Programming

Answer Set Programming (ASP) [18, 54] is a logic programming paradigm for Knowledge Representation and Reasoning that has become of great importance in the last years to the point that it has been largely exploited in both academic and industrial applications [44]. For the sake of completeness, we include here the basic definitions used in the rest of the thesis.

We start from a finite signature At, a non-empty set of propositional atoms. A *rule* is an implication of the form:

$$p_1 \vee \cdots \vee p_m \leftarrow q_1 \wedge \cdots \wedge q_n \wedge \neg s_1 \wedge \cdots \wedge \neg s_j \wedge \neg\neg t_1 \wedge \cdots \wedge \neg\neg t_k \tag{2.1}$$

Given a rule r of the form 2.1 we refer to the disjunction $p_1 \vee \cdots \vee p_m$ as the *head* of r, written $Head(r)$, and denote the set of head atoms as $H(r) \overset{\text{df}}{=} \{p_1, \ldots, p_m\}$ The conjunction in the antecedent of the implication is called the *body* of r and denoted as $Body(r)$. We also define the positive and negative parts of the body respectively as the conjunctions $Body^+(r) \overset{\text{df}}{=} q_1 \wedge \cdots \wedge q_n$ and $Body^-(r) \overset{\text{df}}{=} \neg s_1 \wedge \cdots \wedge \neg s_j \wedge \neg\neg t_1 \wedge \cdots \wedge \neg\neg t_k$. The atoms in the positive body are represented as $B^+(r) \overset{\text{df}}{=} \{q_1, \ldots, q_n\}$. As usual, an empty disjunction (resp. conjunction) stands for $\bot$ (resp. $\top$). A rule r with empty head $H(r) = \emptyset$ is called a *constraint*. On the other hand, when $H(r) = \{p\}$ is a singleton, $B^+(r) = \emptyset$ and $Body^-(r) = \top$ the rule has the form $p \leftarrow \top$ and is said to be a *fact*, simply written as p. The use of double negation in the body allows for representing elementary choice rules. For instance, we will sometimes use the abbreviation $\{p\} \leftarrow B$ to stand for $p \leftarrow B \wedge \neg\neg p$.

An ASP program P is a set of rules. A program P is *positive* if $Body^-(r) = \top$ for all rules $r \in P$. A program P is *non-disjunctive* if $|H(r)| \leq 1$ for every rule $r \in P$.

Definition 3 (Horn Program)**.** *A program P is said to be* Horn *if it is both positive and non-disjunctive. That is, if for every rule $r \in P$, $Body^-(r) = \emptyset$ and $|Head(r)| <= 1$*

Note that the condition $|Head(r)| <= 1$ implies that a Horn program may include (positive) constraints $\bot \leftarrow B$.

A propositional interpretation I is any subset of atoms $I \subseteq At$. We say that a propositional interpretation is a (classical) model of a program P if $I \models \mathit{Body}(r) \rightarrow \mathit{Head}(r)$ in classical logic, for every rule $r \in P$.

Definition 4 (Reduct). *Given a program P and a set of atoms M, the* Reduct *of P w.r.t M, written P^M is the program resulting of:*

1. *removing every rule of the form 2.1 such that some $s_i \in M$,*
2. *removing every remaining negative literal in the program.*

Definition 5 (Stable Model). *Given an interpretation I, we say it is a* stable model *(or* answer set*) of a program P, if I is a minimal model of P^I. We denote the set of all stable models of P as $SM(P)$*

2.1.1 Monotonicity vs Non-Monotonicity

Monotonicity is a property that many logical systems possess, such as, for instance, classical logic. Given three different theories in classical logic T, γ and σ monotonicity states that, if $T \models \gamma$, then $T \cup \sigma \models \gamma$, for any possible theory σ. In other words, if something is entailed by a theory, the addition of new knowledge cannot change that.

Non-monotonicity breaks that property. Under a non-monotonic setting such as ASP, the inclusion of new knowledge can lead to different conclusions. For instance, take the following program:

$$wet \leftarrow not\ battery. \tag{2.2}$$

This program has only one answer set, namely $A_1 = \{wet\}$, so wet is entailed by the rule 2.2. However, by the addition of the rule $battery \leftarrow \top$, A_1 is no longer an answer set of the new program whose unique answer set now is $A_2 = \{battery\}$, so wet is no longer entailed.

2.1.2 Grounding Answer Set Programs With Variables

ASP systems allow the use of predicates. That is, rules may have atoms of the form $atom(t_1, t_2, \ldots, t_n)$ where t_i are terms (that is either contents of variables). A variable is always represented as an identifier starting with a capital letter such as X, Y, Z, etc. An atom (or literal) having any variable is referred to as *non-ground*, and *ground* otherwise. If a rule has any non-ground atom or literal, then the rule is also non-ground (ground otherwise). Before finding the answer sets of a program, most ASP systems first perform a grounding step over the rules of the program. Take for instance the following program P_3,

$$wireless(1). \tag{2.3}$$
$$wireless(2). \tag{2.4}$$
$$up(X) : -wireless(X). \tag{2.5}$$

where rules 2.3 and 2.4 are ground facts, and 2.5 is a non-ground rule. Intuitively, the process of *grounding* is a replacement of a program P by another $ground(P)$ where variables are replaced by all their possible instantiations, that is, by the possible constants (from those occurring in P). As an example, take program $ground(P_3)$ corresponds to:

$$wireless(1). \tag{2.6}$$
$$wireless(2). \tag{2.7}$$
$$up(1) :- wireless(1). \tag{2.8}$$
$$up(2) :- wireless(2). \tag{2.9}$$

2.1.3 Aggregates

An *aggregate element* has the form:

$$t_1, \dots, t_m : l_1, \dots, l_n \tag{2.10}$$

where t_i are *terms* (i.e. constants, numerical terms, function terms, or variables) and l_i are literals. Then an *aggregate atom* is defined as:

$$\#aggr\ \{E\} \prec u \tag{2.11}$$

where $\#aggr \in \{\#max, \#min, \#sum, \#count\}$ is an *aggregate function name*, $\prec \in \{<, \leq, >, \geq, =, \neq\}$ is an *aggregate relation*, u is a term, and E is a possibly infinite collection of aggregate elements separated by the symbol ";". Being a an aggregate atom, a and `not` a are aggreagte literals.

What is special about aggregates is that they can be used as literals in the body of rules and thus have their own truth value, but also act as *functions* computing (and retrieving) values. Each different aggregate function (name) has a different interpretation. For instance the rule

$$old\_students(N) \leftarrow \#count\{S : student(S), age(S, A), A > 65\} = N. \tag{2.12}$$

contains a $\#count$ aggregate that for any pair of values S, A such there exists some $student(S)$, $age(S, A)$ atoms that are true. Whenever the aggregate atom holds, then it does so with an associated value. In rule 2.12, variable N captures that value whenever the aggregate literal is true, therefore the rule is supported and an atom $old_stu dents(N)$ (for a particular value for N) is derived.

Consider this other example

$$person(p). \tag{2.13}$$
$$property(p, 12). \tag{2.14}$$
$$property(p, 10). \tag{2.15}$$
$$grant(P) \leftarrow \#sum\{C : property(P, C)\} < 25, person(P). \tag{2.16}$$

Here, two $property(P, C)$ facts are being taken into account in the aggregate. Each value C such that $property(P, C)$ is true is being added to the aggregate so that the total is then checked to be lower than 25, which is true, thus the rule is satisfied and the atom $grant(p)$ is derived.

An aggregate atom is said to be *monotone* when the increase of any of its arguments keeps the atom true, if it was true before. For instance, the $\#sum$ aggregate in 2.16 is not monotone: increasing 12 to 15 makes the atom become false. On the other hand, changing '<' by '>' would make the aggregate become monotone.

2.1.4 Choice rules

A choice element is an element of the form:

$$a : l_1, \ldots, l_k \tag{2.17}$$

where a is a classical atom and $l_1, \ldots, l_k$ are literals. Then a choice atom has the form:

$$\{C\} \prec u \tag{2.18}$$

where C is a collection of choice elements separated by ";", $\prec \in \{<, \leq, >, \geq, =, \neq\}$ and u is a term (i.e. a constant, arithmetic term, or a variable). $\prec$ and u defaults to $\geq$ and 0 respectively.

Finally, choice rules have the form:

$$C \prec u \leftarrow b_1, \ldots, b_n. \tag{2.19}$$

where $C \prec u$ is a choice atom and $b_1, \ldots, b_n$ are literals. Intuitively, when the body of a choice rule is satisfied, instead of deriving an atom, we may derive any subset of a atoms belonging to the choice elements in C, such that its corresponding literals $l_1, \ldots, l_k$ are also true. Furthermore, the number of derived atoms must comply with the condition imposed by $\prec u$.

To illustrate how choice rules work, consider program P_4,

$$\begin{aligned}
person(p). \quad & (2.20)\\
shirt(blue). \quad & (2.21)\\
shirt(white). \quad & (2.22)\\
shirt(pink). \quad & (2.23)\\
shirt(green). \quad & (2.24)\\
dirty(green). \quad & (2.25)\\
\{packed(S) : shirt(S), not\ dirty(S)\} <= 2 : -person(p). \quad & (2.26)
\end{aligned}$$

As atom $person(p)$ is true by fact 2.20, the choice rule 2.26 is satisfied and thus we have to consider the different subsets of atoms that will be derived. First, as the green shirt is dirt (fact 2.25) the choice atom is not true when $S = green$ and thus $packed(green)$ cannot be part of any set of derivated atoms. Then the possible derivable subsets are comprising 2 or fewer atoms in $S = \{packed(blue), packed(white), packed(pink)\}$. Those are any $S' \subseteq S$ such that $|S'| <= 2$

$$\begin{aligned} S' = \{&\emptyset, \\ &\{packed(blue)\}, \\ &\{packed(white)\}, \\ &\{packed(pink)\}, \\ &\{packed(blue), packed(white)\}, \\ &\{packed(blue), packed(pink)\}, \\ &\{packed(white), packed(pink)\}\} \end{aligned}$$

2.2 ASP Solver Clingo and Python API

2.2.1 Pooling

On top of standard ASP expressions, `clingo` defines its own extensions, enriching the language. *Pooling* is an example of a shortcut to enhance the expressivity and compactness of ASP programs. By the use of the symbol ; in the head (or in some literal in the body) of a rule, one can save space and group several rule definitions in one. The intuition is that the rules break into several rules. For instance, consider the following definition using pooling.

```
shirt(blue;white;pink;green).
```

Between each possible value, we use the symbol ; here splits the definition of independents Creating a different fact for each different value. The below's program below is equivalent to the previous one.

```
shirt(blue). shirt(white). shirt(pink). shirt(green).
```

2.2.2 #show and #project

The `#show` directive defines the set of atoms that are included in the output when visualizing the answer sets of a program. To illustrate how it works, consider the following program:

```
shirt(blue;white;pink;green).
trousers(jeans;fabric).

2 {selected(S): shirt(S); selected(P): trousers(P)} 2.
```

where exactly two items of clothing are chosen from a set of shirts and trousers. The answer sets of the program are listed below.

```
Answer: 1
shirt(blue) shirt(white) shirt(pink) shirt(green) trousers(jeans) trousers(fabric) selected(fabric) selected(white)
Answer: 2
shirt(blue) shirt(white) shirt(pink) shirt(green) trousers(jeans) trousers(fabric) selected(fabric) selected(blue)
Answer: 3
shirt(blue) shirt(white) shirt(pink) shirt(green) trousers(jeans) trousers(fabric) selected(fabric) selected(green)
    ...
```

Then the #show directive can be used to specify a particular set of atoms to be shown, The first option is to specify predicates that we want to list in the output. For instance by introducing `#show sentence/1.` into the program we obtain the output below.

```
Answer: 1
selected(fabric) selected(white)
Answer: 2
selected(fabric) selected(blue)
Answer: 3
selected(fabric) selected(green)
        ...
```

Additionally, the #show can be used to define a precise set of atoms to be visualized. This is done by using it as a conditional atom. Consider the following variation of the program

```
shirt(blue;white;pink;green).
trousers(jeans;fabric).

2 {selected(S): shirt(S); selected(P): trousers(P)} 2.

#show.
#show selected(S) : selected(S), trousers(S).
```

The first #show declarative filters out every atom in the answer sets. The second defines a set of atoms `selected(S)` such that the conjunction `selected(S), trousers(S)` is true. The atoms in the defined set are those that will be shown in the answer sets. The next output lists the corresponding output.

```
Answer: 1
selected(fabric)
Answer: 2
selected(jeans)
Answer: 3
selected(jeans) selected(fabric)
Answer: 4
selected(jeans)
    ...
```

Additionally, we can also make use of #project to collapse several answer sets. When we use #show as in the previous example it could happen that several answer sets show the exact same atoms. For instance, in the last output we have shown, Answer 2 and Answer 4 show the same atoms. Of course, they do not represent the same answer set, but because of our previously defined #show, they collapse to the same relevant atoms. For instance, by simply including the option `-project` we avoid this behavior (see the output listed below).

```
Answer: 1
selected(fabric)
Answer: 2

Answer: 3
selected(jeans) selected(fabric)
Answer: 4
selected(jeans)
```

As we are only showing the selected trousers and we collapse the answer sets in terms of these atoms, we only get four answers, one corresponding to each possible combination. Moreover #project can be used in a similar manner than #show to specify the set of atoms you want to perform the collapse on.

2.2.3 `clingo`'s Abstract Syntax Tree

On top of being a command line tool `clingo` can further be used as an API [1] controlled by a procedural language such as Python. it provides much more functionality and control over the operation of the ASP solver than from the usual Command Line Tool (CLI). In addition to allowing greater control of the solver and grounder, it is also possible to access other functionalities. One of particular importance is the possibility of accessing the *Abstract Syntax Tree* (AST).

When parsing an ASP program, each sentence (i.e. each ASP language construct ending with a point .) is processed and a Python object corresponding to the sentence is created. Each element of the grammar [2] of `clingo` has its own, unique Python class. Although in terms of functionality, the different classes behave almost the same, identifying the different grammar elements allows us to analyze the syntax tree of a program. The objects are organized in a tree-like structure such that more general grammar elements are father nodes of the more elemental ones For instance, when `clingo` parses a rule it creates an instance of the `ASTType.Rule` class. The child of this object can be inspected to find the head and the body of the rule, each one implementing a class representing its corresponding element in the grammar.

One functionality that they offer is the use of `Transofmers`. The transformers are an abstract class offered by the `clingo`'s Python API. The idea behind it is that engineers can define their own Python classes by implementing a transformer so that, during parsing, their code intervenes just when certain grammar elements are parsed. The word transformer comes from the notion that this can be used to modify (thus, transform) the input program before grounding (during parsing). For instance, Figure 2.1 shows an example using a transformer for renaming variables of the program.

```python
from clingo.ast import Transformer, Variable, parse_string

class VariableRenamer(Transformer):
    def visit_Variable(self, node):
        return node.update(name='_' + node.name)

vrt = VariableRenamer()
parse_string('p(X) :- q(X).', lambda stm: print(str(vrt(stm))))

# Output:
#  p(_X) :- q(_X).
```

Figure 2.1: Example of Python code defining a transformer for renaming the variables of the input logic program.

The code defines a `Transformer` (child) class called `VariableRenamer`. In it, provides a method `visit_Variable`, whose specific naming is not arbitrary but responds to a pattern such that `visit_<GrammarElement>`, meaning that the method will be invoked whenever a variable is parsed. When parsing the program

[1]https://potassco.org/clingo/python-api/5.6/clingo/
[2]https://potassco.org/clingo/python-api/5.6/clingo/ast.html

$p(X) \leftarrow q(X)$. in line 8, the method is invoked twice (one for each variable `X` in the rule), and updates the names of the variables. In line 11, the result is shown as a commented line.

2.2.4 Executing Python code within `clingo` with Context class

If the `clingo` installation is set up correctly, it will be able to execute Python code embedded in logic programs. This is done by providing `clingo` with a special object called *Context*. Users can write their custom Context classes, and provide them with methods that can be called during the instantiation (grounding) process. This methods are called directly in the ASP code by using the ‘’ symbol.

In the case of `xclingo`, this is used to replace the placeholders in the text annotations with the corresponding variable values. Figure 2.2 shows the Python class implementing this functionality. When called, the function takes an arbitrary `text` that may contain some placeholders ‘`%`’ and a tuple

```
class XClingoContext:
"""Xclingo context class."""

def label(self, text, tup):
    """Given the text of a label and a tuple of symbols, handles the variable instantiation
    and returns the processed text label."""
    if text.type == SymbolType.String:
        text = text.string
    else:
        text = str(text).strip('"')
    for val in tup.arguments:
        text = text.replace("%", val.string if val.type == SymbolType.String else str(val), 1)
    return [String(text)]

```

Figure 2.2: A Context class for binding variable values to text placeholders.

of values `tup` and orderly replaces each appearance of a placeholder by its respective value in the tuple. Figure 2.3 shows a small program that can be executed by `clingo` and that uses that Context class. Between lines 1 and 22, enclosed by `#script (python)` and `#end.`, we define a Python code block, where we define our Context class and we modify the main behavior of `clingo` to introduce our defined functionality. The last two lines are the ASP rules for our program. When solving the program with `clingo`, the method `label` will be called when instantiating the second rule (the one with the atom `some_text` in the head). Within that call to the function, the value for the `text` variable would be the string `"% is sentenced to %"` while the value for the `tup` variable would be (in this particular case) `(gabriel, prison)`, meaning that variable `P` took the value `gabriel` and variable `S` took the value `prison`. The method returns the string after making the replacements. Output 2.1 shows the result of solving the program in Figure 2.3.

```
#script (python)
from clingo import String
from clingo.symbol import SymbolType

class Context:
    """Xclingo context class."""

    def label(self, text, tup):
        """Given the text of a label and a tuple of symbols, handles the variable instantiation
        and returns the processed text label."""
        if text.type == SymbolType.String:
            text = text.string
        else:
            text = str(text).strip('"')
        for val in tup.arguments:
            text = text.replace("%", val.string if val.type == SymbolType.String else str(val), 1)
        return [String(text)]

def main(prg):
    prg.ground([("base", [])], context=Context())
    prg.solve()
#end.

sentence(gabriel, prison).
some_text(@label("% is sentenced to %", (P, S,))) :- sentence(P, S).
```

Figure 2.3: Small program using the context Class from Figure 2.2

```
Answer: 1
sentence(gabriel,prison) some_text("gabriel is sentenced to prison")
SATISFIABLE
```

Output 2.1: Output after solving the program in Figure 2.3 with clingo.

2.2.5 Theory definitions

Part of `clingo` language syntax is devoted to theory specification [53]. This functionality aims to provide `clingo` users with an easy way to extend the solver's functionality without the need to manually modify any of its parts.

First, users have to provide some *theory definitions*, which have the form

$$\#theory\, T\{D_1 : \ldots ; D_n\}.$$

where T is the theory name and each D_i is a definition for a *theory term* or a *theory atom*. A theory atom has the form

$$\&p/k\ :\ t, o \quad or \quad \&p/k\ :\ t, \{\diamondsuit_1, \ldots, \diamondsuit_m\}, t', o$$

where p is a predicate name of arity k, t and t' are names of some theory term definitions, each $\diamondsuit_i$ is a *theory operator* for $m \geq 1$, and $o \in \{head, body, any, directive\}$ determines where the theory atom may occur in a rule. Finally, a theory operator has the form

$$\diamondsuit\ :\ p, unary \quad or \quad \diamondsuit\ :\ p, binary, a$$

where $\diamondsuit$ is a unary or binary theory operator with precedence $p \geq 0$. Binary operators' associativity is given by $a \in \{right, left\}$.

By providing these definitions, one can then include those constructs into the logic program, so that they will become accepted as part of the syntax. The theory atoms used in the program will become part of the AST representation of the program, and thus they can be manipulated. This can be used to integrate all kinds of expressions adapting the expressivity of the language to a specific problem. Also, when using the `clingo` Python API special behavior can be defined for these special atoms, extending the functionality of the solver.

In the case of `xclingo`, they are used to extend `clingo` syntax to include a new set of annotations. We use these annotations to allow the user to create natural language explanations. These annotations are handled internally as theory atoms, and thus they are easily manipulated by `xclingo` as another member of the program in the AST.

Part I

Explanation in Answer Set Programming

Chapter 3

Support Graphs

3.1 Introduction

In this Chapter, we describe a formal characterization of explanations in terms of graphs constructed with atoms and program rule labels. Under this framework, models may be *justified*, meaning that they have one or more *support graphs*, or *unjustified* otherwise. We prove that all stable models are justified whereas, in general, the opposite does not hold, at least for disjunctive programs. We also characterize a pair of basic operations on graphs, which we call edge pruning and node forgetting, that allow performing information filtering in the explanations. These formal definitions constitute the basis of `xclingo`, which relies on an ASP encoding to generate the explanation graphs of a given answer set of some original program. We prove the soundness and correctness of this encoding and then proceed to explain the new `xclingo` specification language, in terms of the effects it produces on support graphs.

The rest of this chapter is structured as follows. Section 3.2 provides the definitions for this framework. Section 3.3 defines two operations over the explanation graphs used to remove irrelevant information from the graphs. Section 3.4 describes a simple ASP encoding for computing the graphs and proves its soundness and completeness.

3.2 Explanations as Support Graphs

We start from a finite[1] signature At, a non-empty set of propositional atoms. A *(labelled) rule* is an implication of the form:

$$\ell : p_1 \vee \cdots \vee p_m \leftarrow q_1 \wedge \cdots \wedge q_n \wedge \neg s_1 \wedge \cdots \wedge \neg s_j \wedge \neg\neg t_1 \wedge \cdots \wedge \neg\neg t_k \tag{3.1}$$

Given a rule r like (3.1), we denote its label as $Lb(r) \overset{\text{df}}{=} \ell$. We also call the disjunction in the consequent $p_1 \vee \cdots \vee p_m$ the *head* of r, written $Head(r)$, and denote the set of head atoms as $H(r) \overset{\text{df}}{=} \{p_1, \ldots, p_m\}$; the conjunction in the antecedent is called the *body* of r and denoted as $Body(r)$. We

[1]We leave the study of infinite signatures for future work. This will imply explanations of infinite size, but each one should contain a finite proof for each atom.

also define the positive and negative parts of the body respectively as the conjunctions $Body^+(r) \overset{\text{df}}{=} q_1 \wedge \cdots \wedge q_n$ and $Body^-(r) \overset{\text{df}}{=} \neg s_1 \wedge \cdots \wedge \neg s_j \wedge \neg\neg t_1 \wedge \cdots \wedge \neg\neg t_k$. The atoms in the positive body are represented as $B^+(r) \overset{\text{df}}{=} \{q_1, \ldots, q_n\}$. As usual, an empty disjunction (resp. conjunction) stands for $\bot$ (resp. $\top$). A rule r with empty head $H(r) = \emptyset$ is called a *constraint*. On the other hand, when $H(r) = \{p\}$ is a singleton, $B^+(r) = \emptyset$ and $Body^-(r) = \top$ the rule has the form $\ell : p \leftarrow \top$ and is said to be a *fact*, simply written as $\ell : p$. The use of double negation in the body allows representing elementary choice rules. For instance, we will sometimes use the abbreviation $\ell : \{p\} \leftarrow B$ to stand for $\ell : p \leftarrow B \wedge \neg\neg p$. A *(labelled) logic program* P is a set of labelled rules where no label is repeated. Note that P may still contain two rules r, r' with same body and head $Body(r) = Body(r')$ and $H(r) = H(r')$, but different labels $Lb(r) \neq Lb(r')$. A program P is *positive* if $Body^-(r) = \top$ for all rules $r \in P$. A program P is *non-disjunctive* if $|H(r)| \leq 1$ for every rule $r \in P$. Finally, P is *Horn* if it is both positive and non-disjunctive: note that this may include (positive) constraints $\bot \leftarrow B$.

A propositional interpretation I is any subset of atoms $I \subseteq At$. We say that a propositional interpretation is a (classical) model of a labelled program P if $I \models Body(r) \rightarrow Head(r)$ in classical logic, for every rule $r \in P$. The *reduct* of a labelled program P with respect to I, written P^I, is a simple extension of the standard reduct by [57] that collects now the *labelled* positive rules:

$$P^I \overset{\text{df}}{=} \{\, Lb(r) : Head(r) \leftarrow Body^+(r) \mid r \in P,\ I \models Body^-(r) \,\}$$

As usual, an interpretation I is a *stable model* (or *answer set*) of a program P if I is a minimal model of P^I. Note that, for the definition of stable models, the rule labels are irrelevant. We write $SM(P)$ to stand for the set of stable models of P.

We define the rules of a program P that *support* an atom p under interpretation I as $SUP(P, I, p) \overset{\text{df}}{=} \{r \in P \mid p \in H(r), I \models Body(r)\}$ that is, rules with p in the head whose body is true with respect to I. The next proposition proves that, given I, the rules that support p in the reduct P^I are precisely the positive parts of the rules that support p in P.

Proposition 1. *For any model $I \models P$ of a program P and any atom $p \in I$: $SUP(P^I, I, p) = SUP(P, I, p)^I$.*

Proof. We prove first $\supseteq$: suppose $r \in SUP(P, I, p)$ and let us call $r' = Lb(r) : Head(r) \leftarrow Body^+(r)$. Then, by definition, $I \models Body(r)$ and, in particular, $I \models Body^-(r)$, so we conclude $r' \in P^I$. To see that $r' \in SUP(P^I, I, p)$, note that $I \models Body(r)$ implies $I \models Body^+(r) = Body(r')$.

For the $\subseteq$ direction, take any $r' \in SUP(P^I, I, p)$. By definition of reduct, we know that r' is a positive rule and that there exists some $r \in P$ where $Lb(r) = Lb(r')$, $H(r) = H(r')$, $B^+(r) = B^+(r')$ and $I \models Body^-(r)$. Consider any rule r satisfying that condition (we could have more than one): we will prove that $r \in SUP(P, I, p)$. Since $r' \in SUP(P^I, I, p)$, we get $I \models Body(r')$ but this is equivalent to $I \models Body^+(r)$. As we had $I \models Body^-(r)$, we conclude $I \models Body(r)$ and so r is supported in P given I. □

Definition 6 (Support Graph/Explanation)**.** *Let P be a labelled program and I a classical model of P. A* support graph *G of I under P is a labelled directed graph $G = \langle I, E, \lambda\rangle$ whose vertices are the* *atoms in I, the edges in $E \subseteq I \times I$ connect pairs of atoms, the total function $\lambda : I \rightarrow Lb(P)$ assigns a label to each atom, and G further satisfies:*

(i) *λ is injective*

(ii) *for every $p \in I$, the rule r such that $Lb(r) = \lambda(p)$ satisfies:*
$r \in SUP(P, I, p)$ and $B^+(r) = \{q \mid (q, p) \in E\}$.

A support graph G is said to be an explanation *if it additionally satisfies:*

(iii) *G is acyclic.* □

Condition (i) means that there are no repeated labels in the graph, i.e., $\lambda(p) \neq \lambda(q)$ for different atoms $p, q \in I$. Condition (ii) requires that each atom p in the graph is assigned the label ℓ of some rule with p in the head, with a body satisfied by I and whose atoms in the positive body form all the incoming edges for p in the graph. Intuitively, labelling p with ℓ means that the corresponding (positive part of the) rule has been fired, "producing" p as a result. Since a label cannot be repeated in the graph, each rule can only be used to produce one atom, even though the rule head may contain more than one (when it is a disjunction). In general, program P may allow alternative ways of deriving an atom p in a model I. Thus, a same model I may have multiple support graphs under P, as we will illustrate later.

It is not difficult to see that an explanation $G = \langle I, E, \lambda \rangle$ for a model I is uniquely determined by its atom labelling λ. This is because condition (ii) about λ in Definition 6 uniquely specifies all the incoming edges for all the nodes in the graph. On the other hand, of course, not every arbitrary atom labelling corresponds to a well-formed explanation. We will sometimes abbreviate an explanation G for a model I by just using its labelling λ represented as a set of pairs of the form $\lambda(p) : p$ with $p \in I$.

Definition 7 (Supported/Justified model)**.** *A classical model I of a labelled program P if $I \models P$ is said to be a* supported model *of P if there exists some support graph of I under P. Moreover, I is said to be a* justified model *of P if there exists some explanation G (i.e. acyclic support graph) of I under P. We write $SPM(P)$ and $JM(P)$ to respectively stand for the set of supported and justified models of P.* □

The name of *supported model* is not casual: we prove later on that, for non-disjunctive programs, the above definition coincides with the traditional one in terms of fixpoints of the immediate consequences operator [42] or as models of Clark's completion [26].

From Definition 7, it is clear that all justified models are obviously supported $JM(P) \subseteq SPM(P)$ but, in general, the opposite does not hold, as we will see later. Our main focus, however, is on justified models, since we will relate them to proofs, that are always acyclic. We can observe that not all models are justified, whereas a justified model may have more than one explanation, as we illustrate next.

Example 2. *Consider the labelled logic program P*

$$\ell_1 : a \vee b \qquad \ell_2 : d \leftarrow a \wedge \neg c \qquad \ell_3 : d \leftarrow \neg b$$

No model $I \models P$ with $c \in I$ is justified since c does not occur in any head, so its support is always empty $SUP(P, I, c) = \emptyset$ and c cannot be labelled. The models of P without c are $\{b\}$, $\{a, d\}$, $\{b, d\}$ and

$\{a, b, d\}$ but only the first two are justified. The explanation for $I = \{b\}$ corresponds to the labelling $\{(\ell_1 : b)\}$ (it forms a graph with a single node). Model $I = \{a, d\}$ has the two possible explanations:

$$\ell_1 : a \longrightarrow \ell_2 : d \qquad\qquad \ell_1 : a \quad \ell_3 : d \tag{3.2}$$

Model $I = \{b, d\}$ is not justified: we have no support for d given I, $SUP(P, I, d) = \emptyset$, because I satisfies neither bodies of ℓ_2 nor ℓ_3. On the other hand, model $\{a, b, d\}$ is not justified either, because $SUP(P, I, a) = SUP(P, I, b) = \{\ell_1\}$ and we cannot use the same label ℓ_1 for two different atoms a and b in a same explanation (condition (i) in Def. 6). □

Definition 8 (Proof of an atom)**.** *Let I be a model of a labelled program P, $G = \langle I, E, \lambda\rangle$ an explanation for I under P and let $p \in I$. The* proof *for p induced by G, written $\pi_G(p)$, is the derivation:*

$$\pi_G(p) \overset{df}{=} \frac{\pi_G(q_1)\ \dots\ \pi_G(q_n)}{p}\ \lambda(p),$$

where, if $r \in P$ is the rule satisfying $Lb(r) = \lambda(p)$, then $\{q_1, \dots, q_n\} = B^+(r)$. When $n = 0$, the derivation antecedent $\pi_G(q_1)\ \dots\ \pi_G(q_n)$ is replaced by $\top$ (corresponding to the empty conjunction). □

Example 3. *Let P be the labelled logic program:*

$$\ell_1 :\ p \qquad\quad \ell_2 :\ q \leftarrow p \qquad\quad \ell_3 :\ r \leftarrow p, q$$

P has a unique justified model $\{p, q, r\}$ whose explanation is shown in Figure 3.1 (left) whereas the induced proof for atom r is shown in Figure 3.1 (right). □

$$\begin{tikzcd}
\ell_1 :\ p \arrow[r] \arrow[rr, bend left] & \ell_2 :\ q \arrow[r] & \ell_3 :\ r
\end{tikzcd}$$

Explanation

$$\dfrac{\dfrac{\dfrac{\top}{p}\,(\ell_1)}{q}\,(\ell_2) \qquad \dfrac{\top}{p}\,(\ell_1)}{r}\,(\ell_3)$$

Proof for atom r

Figure 3.1: Some results for model $\{p, q, r\}$ of program in Example 3.

The next proposition trivially follows from the definition of explanations:

Proposition 2. *If P is a Horn program, and G is an explanation for a model I of P then, for every atom, $p \in I$, $\pi_G(p)$ corresponds to a Modus Ponens derivation of p using the rules in P.*

It is worth mentioning that explanations do not generate any arbitrary Modus Ponens derivation of an atom, but only those that are globally "coherent" in the sense that, if any atom p is repeated in a proof, it is always justified repeating *the same subproof*.

In the previous examples, justified and stable models coincided: one may wonder whether this is a general property. As we see next, however, every stable model is justified but, in general, the opposite may not hold. To prove that stable models are justified, we start proving a correspondence between explanations for any model I of P and explanations under P^I.

Proposition 3. *Let I be a model of program P. Then G is an explanation for I under P iff G is an explanation for I under P^I.*

Proof. By Proposition 1, for any atom $p \in I$, the labels in $SUP(P, I, p)$ and $SUP(P^I, I, p)$ coincide, so there is no difference in the ways in which we can label p in explanations for P and for P^I. On the other hand, the rules in $SUP(P^I, I, p)$ are the positive parts of the rules in $SUP(P, I, p)$, so the graphs we can form are also the same. □

Corollary 1. $I \in JM(P)$ *iff* $I \in JM(P^I)$.

Theorem 1. *Stable models are justified:* $SM(P) \subseteq JM(P)$.

Proof. Let I be a stable model of P. To prove that there is an explanation G for I under P, we can use Proposition 1 and just prove that there is some explanation G for I under P^I. We will build the explanation with a non-deterministic algorithm where, in each step i, we denote the graph G_i as $G_i = \langle I_i, E_i, \lambda_i \rangle$ and represent the labelling λ_i as a set of pairs of the form $(\ell : p)$ meaning $\ell = \lambda(p)$. The algorithm proceeds as follows:

```
1:  $I_0 \leftarrow \emptyset; E_0 \leftarrow \emptyset; \lambda_0 \leftarrow \emptyset$
2:  $G_0 = \langle I_0, E_0, \lambda_0 \rangle$
3:  $i \leftarrow 0$
4:  while $I_i \not\models P^I$ do
5:      Pick a rule $r \in P^I$ s.t. $I_i \models Body(r) \wedge \neg Head(r)$
6:      Pick an atom $p \in I \cap H(r)$
7:      $I_{i+1} \leftarrow I_i \cup \{p\}$
8:      $\lambda_{i+1} \leftarrow \lambda_i \cup \{(\ell : p)\}$
9:      $E_{i+1} \leftarrow E_i \cup \{(q, p) \mid q \in B^+(r)\}$
10:     $G_{i+1} \leftarrow \langle I_i, E_i, \lambda_i \rangle$
11:     $i \leftarrow i + 1$
12: end while
```

The existence of a rule $r \in P^I$ in line 5 is guaranteed because the **while** condition asserts $I_i \not\models P^I$ and so there must be some rule whose positive body is satisfied by I_i but its head is not satisfied. We prove next that the existence of an atom $p \in I \cap Head(r)$ (line 5) is also guaranteed. First, note that the **while** loop maintains the invariant $I_i \subseteq I$, since $I_0 = \emptyset$ and I_i only grows with atoms p (line 7) that belong to I (line 6). Therefore, $I_i \models Body(r)$ implies $I \models Body(r)$, but since $I \models P^I$, we also conclude $I \models r$ and thus $I \models Head(r)$ that is $I \cap H(r) \neq \emptyset$, so we can always pick some atom p in that intersection. Now, note that the algorithm stops because, in each iteration, I_i grows with exactly one atom from I that was not included before, since $I_i \models \neg Head(r)$, and so, this process will stop provided that I is finite. The **while** stops satisfying $I_i \models P^I$ for some value $i = n$. Moreover, $I_n = I$, because otherwise, as $I_i \subseteq I$ is an invariant, we would conclude $I_n \subset I$ and so I would not be a minimal model of P^I, which contradicts that I is a stable model of P. We remain to prove that the final $G_n = \langle I_n, E_n, \lambda_n \rangle$ is a correct explanation for I under P^I. As we said, the atoms in I are the graph nodes $I_n = I$. Second, we can easily see that G_n is acyclic because each iteration adds a new node p and links this node to previous atoms from $B^+(r) \subseteq I_i$ (remember $I_i \models Body(r)$) so no loop can be formed. Third, no rule label can be repeated, because we go always picking a rule r that is new, since it was not satisfied in I_i but becomes satisfied in I_{i+1} (the rule head $Head(r)$ becomes true).

Last, for every $p \in I$, it is not hard to see that the (positive) rule $r \in P^I$ such that $Lb(r) = \lambda_n(p)$ satisfies $p \in H(r)$ and $B^+(r) = \{q \mid (q, p) \in E\}$ by the way in which we picked r and inserted p in I_i, whereas $I \models Body(r)$ because $I_i \models Body(r)$, r is a positive rule and $I_i \subseteq I$. □

As a result, we get $SM(P) \subseteq JM(P) \subseteq SPM(P)$, that is, justified models lay in between stable and supported.

Proposition 4. *If P is a consistent Horn program then it has a unique justified model I that coincides with the least model of P.*

Proof. Since P is Horn and consistent (all constraints are satisfied) its unique stable model is the least model I. By Theorem 1, I is also justified by some explanation G. We remain to prove that I is the unique justified model. Suppose there is another model $J \supset I$ (remember I is the least model) justified by an explanation G and take some atom $p \in J \setminus I$. Then, by Proposition 2, the proof for p induced by G, $\pi_G(p)$, is a Modus Ponens derivation of p using the rules in P. Since Modus Ponens is sound and the derivation starts from facts in the program, this means that p must be satisfied by any model of P, so $p \in I$ and we reach a contradiction. □

In general, the number of explanations for a single justified model can be exponential, even when the program is Horn, and so, has a unique justified and stable model corresponding to the least classical model, as we just proved. As an example[2]:

Example 4 (A chain of firing squads)**.** *Consider the following variation of the classical* Firing Squad Scenario *introduced by [81] for causal counterfactuals (although we do not use it for that purpose here). We have an army distributed in n squads of three soldiers each, a captain and two riflemen for each squad. We place the squads on a sequence of n consecutive hills $i = 0, \dots, n-1$. An unfortunate prisoner is at the last hill $n-1$, and is being aimed at by the last two riflemen. At each hill i, the two riflemen a_i and b_i will fire if their captain c_i gives a signal to fire. But then, captain c_{i+1} will give a signal to fire if she hears a shot from the previous hill i in the distance. Suppose captain c_0 gives a signal to fire. Our logic program would have the form:*

$$\begin{array}{lll} s_0 : signal_0 & a_i : fireA_i \leftarrow signal_i & a'_{i+1} : signal_{i+1} \leftarrow fireA_i \\ & b_i : fireB_i \leftarrow signal_i & b'_{i+1} : signal_{i+1} \leftarrow fireB_i \end{array}$$

for all $i = 0, \dots, n-1$ where we assume (for simplicity) that $signal_n$ represents the death of the prisoner. This program has one stable model (the least model) making true the $3n+1$ atoms occurring in the program. However, this last model has 2^n explanations because to derive $signal_{i+1}$ from level i, we can choose between any of the two rules a'_i or b'_i (corresponding to the two riflemen) in each explanation.
□

In many disjunctive programs, justified and stable models coincide. For instance, the following example is an illustration of a program with disjunction and head cycles.

Example 5. *Let P be the program:*

$$\ell_1 : p \vee q \qquad \ell_2 : q \leftarrow p \qquad \ell_3 : p \leftarrow q$$

[2]This example was already introduced as Program 7.1 by [47] in his PhD dissertation.

This program has one justified model $\{p, q\}$ *that coincides with the unique stable model and has two possible explanations,* $\{(\ell_1 : p), (\ell_2 : q)\}$ *and* $\{(\ell_1 : q), (\ell_3 : p)\}$*.* □

However, in the general case, not every justified model is a stable model: we provide next a simple counterexample. Consider the program P:

$$\ell_1 : a \vee b \qquad\qquad \ell_2 : a \vee c$$

whose classical models are the five interpretations: $\{a\}$, $\{a, c\}$, $\{a, b\}$, $\{b, c\}$ and $\{a, b, c\}$. The last one $\{a, b, c\}$ is not justified, since we would need three different labels and we only have two rules. Each model $\{a, c\}$, $\{a, b\}$, $\{b, c\}$ has a unique explanation corresponding to the atom labellings $\{(\ell_1 : a), (\ell_2 : c)\}$, $\{(\ell_1 : b), (\ell_2 : a)\}$ and $\{(\ell_1 : b), (\ell_2 : c)\}$, respectively. On the other hand, model $\{a\}$ has two possible explanations, corresponding to $\{(\ell_1 : a)\}$ and $\{(\ell_2 : a)\}$. Notice that, in the definition of explanation, there is no need to fire every rule with a true body in I – we are only forced to explain every true atom in I. Note also that only the justified models $\{a\}$ and $\{b, c\}$ are also stable: this is due to the minimality condition imposed by stable models on positive programs, getting rid of the other two justified models $\{a, b\}$ and $\{a, c\}$. The following theorem asserts that, for non-disjunctive programs, every justified model is also stable.

Theorem 2. *If P is a non-disjunctive program, then $SM(P) = JM(P)$.* □

Proof. Given Theorem 1, we must only prove that, for non-disjunctive programs, every justified model is also stable. Let I be a justified model of P. By Proposition 3, we also know that I is a justified model of P^I. P^I is a positive program and is non-disjunctive (since P was non-disjunctive) and so, P is a Horn program. By Proposition 4, we know I is also the *least model* of P^I, which makes it a stable model of P. □

Moreover, for non-disjunctive programs, we can prove that our definition of supported model, coincides with the traditional one in terms of fixpoints of the immediate consequences operator [42]. Given a non-disjunctive program P, let $T_P(I)$ be defined as $\{p \mid r \in P, I \models Body(r), Head(r) = p\}$.

Theorem 3. *If P is a non-disjunctive program, then $I = T_P(I)$ iff $I \in SPM(P)$.* □

Proof. For left to right, suppose $I = T_P(I)$. It is easy to see that this implies $I \models P$. By definition of T_P, for each atom p there exists some rule r with $Head(r) = p$ and $I \models Body(r)$. Let us arbitrarily pick one of those rules r_p for each p. Then we can easily form a support graph where $\lambda(p) = Lb(r_p)$ and assign all the incoming edges for p as (q, p) such that $q \in Body^+(r_p)$.

For right to left, suppose $I \models P$ and there is some support graph G of I under P. We prove both inclusion directions for $I = T_P(I)$. For $\subseteq$, suppose $p \in I$. Then p is a node in G and there is a rule r such that $\lambda(p) = Lb(r)$, $p = Head(r)$ (P is non–disjunctive) and $I \models Body(r)$. But then $p \in T_P(I)$. For $\supseteq$, take any $p \in T_P(I)$ and suppose $p \notin I$. Then, we have at least some rule $r \in P$ with $I \models Body(r)$ and $I \not\models Head(r)(= p)$, something that contradicts $I \models P$. □

To illustrate supported models in the disjunctive case, consider the program:

$$\ell_1 : a \vee b \leftarrow c \qquad\qquad \ell_2 : c \leftarrow b$$

The only justified model of this program is $\emptyset$ which is also stable and supported. Yet, we also obtain a second supported model $\{b, c\}$ that is justified by the (cyclic) support graph with labelling $\{\ell_1 : b, \ell_2 : c\}$.

3.3 Filtered Explanations

In this section, we consider a pair of operations on explanation graphs that allow filtering their information depending on what a final user may consider relevant or not. These two operations are *edge pruning* and *node forgetting*. The idea behind edge pruning is as follows. Consider a proof $\pi_G(p)$ as a reversed tree, with the atom p in the root, and unfolding the tree upwards until we reach the facts that constitute the leaves. When doing this upwards unfolding, we may reach points in the proof where we are not interested in deepening, so we prefer *pruning* the tree at that node. To put an example, imagine that we are solving a planning problem and we have rules to compute $dist(X, Y)$ standing for the minimum distance between each pair of points X, Y in a graph of locations. To explain a given plan, we are not interested in explaining the values of these distances, whose explanations can be long (they depend on path lengths in the graph) and always repeated, since they do not vary along time. Instead, we may prune the proofs at predicate $dist(X, Y)$, declaring that we are *not interested in showing a justification* for these atoms. The effect of pruning $dist(X, Y)$ in the explanations is the same as if distances had been directly provided as input facts, rather than being precomputed using rules.

Formally, we define edge pruning as an operation in the graph.

Definition 9 (Edge pruning)**.** *Let $G = \langle I, E, \lambda\rangle$ be an explanation and let us define the subset of atoms $A \subseteq At$ and the subset of labels $L \subseteq Lb(P)$. Then, the* (edge) pruning *operation on G produces a new graph $prune(G, A, L) \overset{df}{=} \langle I,\ E \setminus E',\ \lambda\rangle$ where:*

$$E' \overset{df}{=} \big\{(p, q) \in E \mid p \in A\big\} \cup \{(p, q) \in E \mid \lambda(q) \in L\} \qquad \square$$

In other words, we remove the outgoing edges for all pruned atoms A and the incoming edges for all nodes q with a pruned label $\lambda(q) \in L$. As an example, Figure 3.2 shows in G_2 the result of pruning G_1 with $A = \{e\}$ and $L = \{\ell_4\}$.

$$\begin{tikzcd}
 & \ell_1 : a \arrow[dl] \arrow[d] & \ell_2 : b \arrow[dl] \\
\ell_3 : c \arrow[dr] & \ell_4 : d \arrow[r] \arrow[d] & \ell_5 : e \arrow[dl] \arrow[d] \\
 & \ell_6 : f & \ell_7 : g
\end{tikzcd}$$

G_1

$$\begin{tikzcd}
 & \ell_1 : a \arrow[dl] & \ell_2 : b \\
\ell_3 : c \arrow[dr] & \ell_4 : d \arrow[r] \arrow[d] & \ell_5 : e \\
 & \ell_6 : f & \ell_7 : g
\end{tikzcd}$$

$G_2 = prune(G_1, \{e\}, \{\ell_4\})$

$$\begin{tikzcd}
 & \ell_1 : a \arrow[dl] \arrow[dd] \arrow[ddr] & \ell_2 : b \arrow[ddl] \arrow[dd] \\
\ell_3 : c \arrow[dr] & & \\
 & \ell_6 : f & \ell_7 : g
\end{tikzcd}$$

$G_3 = forget(G_1, \{d, e\})$

Figure 3.2: Examples of edge pruning and node forgetting.

The intuition for the second operation, *node forgetting*, is to obtain explanations that only consist of atoms and rules considered relevant, removing the irrelevant information.

Definition 10 (Node forgetting). *Let $G = \langle I, E, \lambda\rangle$ be an explanation and let $A \subseteq At$ be a set of atoms to be removed. Then, the* node forgetting *operation on G produces the graph $forget(G, A) \overset{df}{=} \langle I \setminus A,\ E',\ \lambda\rangle$ where E' contains all edges (p_0, p_n) such that there exists a path $(p_0, p_1), (p_1, p_2), \ldots, (p_{n-1}, p_n)$ in E with $n \geq 0$, $\{p_0, p_n\} \cap A = \emptyset$ and $\{p_1, \ldots, p_{n-1}\} \subseteq A$.* □

Note that E' contains all original edges $(p, q) \in E$ for non-removed atoms $\{p, q\} \cap A = \emptyset$, since they correspond to the case where $n = 1$. Graph G_3 in Figure 3.2 is the result of forgetting nodes $A = \{d, e\}$ on G_1.

If we are going to prune some edges and forget some nodes, the order in which we perform both operations produces different results. For instance, take the graph G:

$$\ell_1 : a \longrightarrow \ell_2 : b \longrightarrow \ell_3 : c$$

and suppose we want to both prune atom $A = \{b\}$ and forget the same atom. If we start by pruning, $forget(prune(G, \{b\}, \emptyset), \{b\})$, the final result leaves two disconnected nodes $\ell_1 : a$ and $\ell_3 : c$. If we start instead by forgetting, $prune(forget(G, \{b\}), \{b\}, \emptyset)$, we get

$$\ell_1 : a \longrightarrow \ell_3 : c$$

because the result of pruning after forgetting has no effect since node b does not exist any more. For this reason, in practice, pruning will be performed in the first place (when we still have all the original nodes in the graph) and forgetting afterwards.

Definition 11 (Filtered explanation). *Given a labelled program P, let L be a set of labels $L \subseteq Lb(P)$, and A, F be sets of atoms $A, F \subseteq At$. A labelled graph G is a (L, A, F)*-filtered explanation *for model I under program P if there is some explanation G' of I under P such that $G = forget(prune(G', A, L), F)$.*

It is easy to see that different (unfiltered) explanations may lead to the same (L, A, F)-filtered explanation. For instance, take the program in Example 2 and suppose we want to apply the filtering $(\emptyset, \emptyset, \{d\})$, that is, we just forget atom d. Then, the two explanations (3.2) we obtained for model $\{a, d\}$ in Example 2 collapse into a single filtered one with the unique labelled node $\ell_1 : a$.

It is not hard to see that we may have two different (unfiltered) explanations G' and G'' An important observation is that, if pruning and forgetting are defined as overall operations on a set of explanations, we may get that two different explanations G_1 and G_2 end up producing the same filtered explanation G_3. For this reason, pruning and forgetting not only reduce the information in each explanation but may also significantly reduce the number of different (filtered) explanations.

3.4 An ASP Encoding to Compute and Filter Explanations

In this section, we focus on the computation of explanations for a given stable model. We assume that we use an ASP solver to obtain the answer sets of some program P and that we have some way to label the rules. For instance, we may use the code line number (or another tag specified by the user), followed by the free variables in the rule and some separator. In that way, after grounding, we get a unique identifier for each ground rule.

To explain the answer sets of P we may build the following (non-ground) ASP program $x(P)$ that can be fed with the (reified) true atoms in I to build the ground program $x(P, I)$. As we will prove, the answer sets of $x(P, I)$ are in one-to-one correspondence with the explanations of I. The advantage of this technique is that, rather than collecting all possible explanations in a single shot, something that is too costly for explaining large programs, we can perform regular calls to an ASP solver for $x(P, I)$ to compute one, several or all explanations of I on demand. Besides, this provides a more declarative approach that can be easily extended to cover new features (such as, for instance, minimisation among explanations).

For each rule in P of the form (3.1), $x(P)$ contains the set of rules:

$$\begin{aligned} sup(\ell) &\leftarrow as(q_1) \wedge \cdots \wedge as(q_n) \wedge as(p_i) \wedge \neg as(s_1) \wedge \cdots \wedge \neg as(s_j) && (3.3)\\ &\quad \wedge \neg\neg as(t_1) \wedge \cdots \wedge \neg\neg as(t_k) && (3.4)\\ \{f(\ell, p_i)\} &\leftarrow f(q_1) \wedge \cdots \wedge f(q_n) \wedge as(p_i) \wedge sup(\ell) && (3.5)\\ \bot &\leftarrow f(\ell, p_i) \wedge f(\ell, p_h) && (3.6) \end{aligned}$$

for all $i, h = 1 \ldots m$ and $i \neq h$, and, additionally $x(P)$ contains the rules:

$$\begin{aligned} f(A) &\leftarrow f(L, A) \wedge as(A) && (3.7)\\ \bot &\leftarrow \textit{not}\, f(A) \wedge as(A) && (3.8)\\ \bot &\leftarrow f(L, A) \wedge f(L', A) \wedge L \neq L' \wedge as(A) && (3.9) \end{aligned}$$

As we can see, $x(P)$ reifies atoms in P using three predicates: $as(A)$ which means that atom A is in the answer set I, so it is an initial assumption; $f(L, A)$ means that rule with label L has been "fired" for atom A, that is, $\lambda(A) = L$; and, finally, $f(A)$ that just means that there exists some fired rule for A or, in other words, we were able to derive A. Predicate $sup(\ell)$ tells us that the body of the rule r with label ℓ is "supported" by I, that is, $I \models Body(r)$. Given any answer set I of P, we define the program $x(P, I) \overset{\text{df}}{=} x(P) \cup \{as(A) \mid A \in I\}$. It is easy to see that $x(P, I)$ becomes equivalent to the ground program containing the following rules:

$$\begin{aligned} \{f(\ell, p)\} \leftarrow f(q_1) \wedge \cdots \wedge f(q_n) \quad & \text{for each rule } r \in P \text{ of the form of (3.1),}\\ & I \models Body(r), p \in H(r) \cap I && (3.10)\\ \bot \leftarrow f(\ell, p_i) \wedge f(\ell, p_j) \quad & \text{for each rule } r \in P \text{ of the form of (3.1),}\\ & p_i, p_j \in H(r),\ p_i \neq p_j && (3.11)\\ f(a) \leftarrow f(\ell, a) \quad & \text{for each } a \in I && (3.12)\\ \bot \leftarrow \textit{not}\, f(a) \quad & \text{for each } a \in I && (3.13)\\ \bot \leftarrow f(\ell, a) \wedge f(\ell', a) \quad & \text{for each } a \in I,\ \ell \neq \ell' && (3.14) \end{aligned}$$

Theorem 4 (Soundness). *Let I be an answer set of P. For every answer set J of program $x(P, I)$ there exists a unique explanation $G = \langle I, E, \lambda \rangle$ of I under P such that $\lambda(a) = \ell$ iff $f(\ell, a) \in J$.* $\square$

Proof. We have to prove that J induces a valid explanation G. Let us denote $At(J) \stackrel{\text{df}}{=} \{a \in At \mid f(a) \in J\}$. Since (3.12) is the only rule for $f(a)$, we can apply completion to conclude that $f(a) \in J$ iff $f(\ell, a) \in J$ for some label ℓ. So, the set $At(J)$ contains the set of atoms for which J assigns some label: we will prove that this set coincides with I. We may observe that $I \subseteq At(J)$ because for any $a \in I$ we have the constraint (3.13) forcing $f(a) \in J$. On the other hand, $At(J) \subseteq I$ because the only rules with $f(a)$ in the head are (3.12) and these are only defined for atoms $a \in I$. To sum up, in any answer set J of $x(P, I)$, we derive exactly the original atoms in I, $At(J) = I$ and so, the graph induced by J has exactly one node per atom in I.

Constraint (3.14) guarantees that atoms $f(\ell, a)$ have a functional nature, that is, we never get two different labels for a same atom a. This allows defining the labelling function $\lambda(a) = \ell$ iff $f(\ell, a) \in J$. We remain to prove that conditions (i)-(iii) in Definition 6 hold. Condition (i) requires that λ is injective, something guaranteed by (3.11). Condition (ii) requires that, informally speaking, the labelling for each atom a corresponds to an activated, supported rule for a. That is, if $\lambda(a) = \ell$, or equivalently $f(\ell, a)$, we should be able to build an edge (q, a) for each atom in the positive body of ℓ so that atoms q are among the graph nodes. This is guaranteed by that fact that rule (3.10) is the only one with predicate $f(\ell, a)$ in the head. So, if that ground atom is in J, it is because $f(q_i)$ are also in J i.e. $q_i \in I$, for all atoms in the positive body of rule labelled with ℓ. Note also that (3.10) is such that $I \models Body(r)$, so the rule supports atom p under I, that is, $r \in SUP(P, I, p)$. Let E be the set of edges formed in this way. Condition (iii) requires that the set E of edges forms an acyclic graph. To prove this last condition, consider the reduct program $x(P, I)^J$. The only difference of this program with respect to $x(P, I)$ is that rules (3.10) have now the form:

$$f(\ell, p) \leftarrow f(q_1) \wedge \cdots \wedge f(q_n) \tag{3.15}$$

for each rule $r \in P$ like (3.1), $I \models Body(r), p \in H(r) \cap I$ as before, but additionally $f(\ell, p) \in J$ so the rule is kept in the reduct. Yet, the last condition is irrelevant since $f(\ell, p) \in J$ implies $f(p) \in J$ so $p \in At(J) = I$. Thus, we have exactly one rule (3.15) in $x(P, I)^J$ per each choice (3.10) in $x(P, I)$. Now, since J is an answer set of $x(P, I)$, by monotonicity of constraints, it satisfies (3.11), (3.13) and (3.14) and is an answer set of the rest of the program P' formed by rules (3.15) and (3.12). This means that J is a minimal model of P'. Suppose we have a cycle in E, formed by the (labelled) nodes and edges $(\ell_1 : p_1) \longrightarrow \cdots \longrightarrow (\ell_n : p_n) \longrightarrow (\ell_1 : p_1)$.Take the interpretation $J' = J \setminus \{f(\ell_1, p_1), \ldots, f(\ell_n, p_n), f(p_1), \ldots, f(p_n)\}$. Since J is a minimal for P' there must be some rule (3.15) or (3.11) not satisfied by J'. Suppose J' does not satisfy some rule (3.11) so that $f(a) \notin J'$ but $f(\ell, a) \in J' \subseteq J$. This means we had $f(a) \in J$ since the rule was satisfied by J so a is one of the removed atoms p_i belonging to the cycle. But then $f(\ell, a)$ should have been removed $f(\ell, a) \notin J'$ and we reach a contradiction. Suppose instead that J' does not satisfy some rule (3.15), that is, $f(\ell, p) \notin J'$ and $\{f(q_1), \ldots, f(g_n)\} \subseteq J' \subseteq J$. Again, since the body holds in J, we get $f(\ell, p) \in J$ and so, $f(\ell, p)$ is one of the atoms in the cycle we removed from J'. Yet, since $(\ell : p)$ is in the cycle, there is some incoming edge from some atom in the cycle and, due to the way in which atom labelling is done, this means that this edge must come from some atom q_i with $1 \leq i \leq n$ in the positive body of the rule whose label is ℓ. But, since this atom is in the cycle, this also means that $f(q_i) \notin J'$ and we reach a contradiction. □

Theorem 5 (Completeness)**.** *Let I be an answer set of P. For every explanation $G = \langle I, E, \lambda \rangle$ of I under P there exists a unique answer set J of program $x(P, I)$ where $f(\ell, a) \in J$ iff $\lambda(a) = \ell$ in G.*$\Box$

Proof. Let I be an answer set of P and $G = \langle I, E, \lambda \rangle$ be some explanation for I under P and let us define the interpretation:

$$J := \{f(a) \mid a \in I\} \cup \{f(\ell, a) \mid \lambda(a) = \ell\}$$

We will prove that J is an answer set of $x(P, I)$ or, in other words, that J is a minimal model of $x(P, I)^J$. First, we will note that J satisfies $x(P, I)^J$ rule by rule. For the constraints, J obviously satisfy (3.7) because it contains an atom $f(a)$ for each $a \in I$. We can also see that J satisfies (3.11) because graph G does not contain repeated labels, so we cannot have two different atoms with the same label. The third constraint (3.14) is also satisfied by J because atoms $f(\ell, a)$, $f(\ell', a)$ are obtained from $\lambda(a)$ that is a function that cannot assign two different labels to a same atom a. Satisfaction of (3.11) is guaranteed since the head of this rule $f(a)$ is always some atom $a \in I$ and therefore $f(a) \in J$. For the remaining rule, (3.10), we have two cases. If $f(\ell, p) \not\in J$ then the rule is not included in the reduct and so there is no need to be satisfied. Otherwise, if $f(\ell, p) \in J$ then the rule in the reduct corresponds to (3.15) and is trivially satisfied by J because its only head atom holds in that interpretation. Finally, to prove that J is a minimal model of $x(P, I)^J$, take the derivation tree $\pi_G(a)$ for each atom $a \in I$. Now, construct a new tree π where we replace each atom p in $\pi_G(a)$ by an additional derivation from $f(\ell, p)$ to $f(p)$ through rule (3.12). It is easy to see that π constitutes a Modus Ponens proof for $f(a)$ under the Horn program $x(P, I)^J$ and the same reasoning can be applied to atom $f(\ell, a) \in J$ that is derived in the tree π for $f(a)$. Therefore, all atoms in J must be included in any model of $x(P, I)^J$. $\Box$

We have proved how the answer sets of program $x(P, I)$ coincide exactly with each explanation G of a model I under program P. We will now show how we can introduce the *Edge pruning* and *Node forgetting* operations (respectively defined in 9 and 10). Let $AP \subseteq I$ be the set of atoms to be pruned, let $LP \subseteq Lb(()P)$ the set of labels to be pruned and $AF \subseteq I$ the set of atoms to be forgot. Recall also program $x(P)$ to define the following non ground program $g(P) \overset{\text{df}}{=} x(p) \cup \{as(A) \mid A \in$

$I\} \cup \{prune(A) \mid A \in AP\} \cup \{prunel(L) \mid L \in LP\} \cup \{keep(A) \mid A \in I \setminus AF\} \cup ER$, where ER represents the following set of rules,

$$\begin{aligned} e(\mathit{Effect}, \mathit{Cause}) \leftarrow\ & f(\mathit{Label}, \mathit{Effect}), \mathit{sup}(\mathit{Label}), f(\mathit{Cause}), \\ & \mathit{not}\ \mathit{prune}(\mathit{Effect}), \\ & f(\mathit{Label}, \mathit{Cause}), \mathit{not}\ \mathit{prunel}(\mathit{Label}) \end{aligned} \tag{3.16}$$

$$\mathit{skip}(A, B) \leftarrow e(A, B), \mathit{not}\ \mathit{keep}(A) \tag{3.17}$$
$$\mathit{skip}(A, B) \leftarrow e(A, B), \mathit{not}\ \mathit{keep}(B) \tag{3.18}$$
$$\mathit{gedge}(\mathit{Effect}, \mathit{Cause}) \leftarrow e(\mathit{Effect}, \mathit{Cause}), \mathit{not}\ \mathit{skip}(\mathit{Effect}, \mathit{Cause}) \tag{3.19}$$

$$\mathit{reach}(A, B) \leftarrow \mathit{skip}(A, B) \tag{3.20}$$
$$\mathit{reach}(A, B) \leftarrow \mathit{reach}(A, Y), \mathit{skip}(Y, B), \mathit{not}\ \mathit{keep}(Y) \tag{3.21}$$
$$\mathit{gedge}(\mathit{Effect}, \mathit{Cause}) \leftarrow \mathit{reach}(\mathit{Effect}, \mathit{Cause}), \mathit{keep}(\mathit{Effect}), \mathit{keep}(\mathit{Cause}) \tag{3.22}$$

$$\mathit{gnode}(A) \leftarrow \mathit{keep}(A) \tag{3.23}$$

When solved, the program $g(P)$ will compute the filtered explanations for model I under P as defined in definition 11, given the sets AP, LP and AF.

The atoms $e(\mathit{Effect}, \mathit{Cause})$ derivated from the head of the rule (3.16), represent the edges of the graph, after the pruning but before the forgetting. As explained in Definition 9, this operation involves deleting two sets of edges. On one hand, the outcoming edges of the atoms in AP represented by the predicate $prune(A)$ are removed, which is handled by the second line of the body of this rule. On the other hand, the same happens for the incoming edges of the atoms labeled with the labels in LP represented by the predicate $prunel(L)$, which is handled by the third line of the body of this rule. Recalling the forgetting operation from Definition 10, it first involves the removal of any edge incident to the atoms on AF (i.e $\mathit{not}\ \mathit{keep}(A)$). Such removals are considered in predicate $\mathit{skip}(\mathit{Effect}, \mathit{Cause})$ computed from rules (3.17) and (3.19).

Predicate $\mathit{gedge}(\mathit{Effect}, \mathit{Cause})$ represents the final edges of the filtered graph (i.e. rule (3.19) means that any non-removed or pruned edge remains). Furthermore, forgetting also involves the creation of new edges connecting nodes that were transitively connected through forgotten atoms. This is modeled by rule (3.22), with the help of the $\mathit{reach}(A, B)$ predicate, defined by rules (3.20) and (3.21), which means that you can reach from non-forgotten node A to non-forgotten node B traversing only through forgotten nodes and edges. Finally, predicate $\mathit{gnode}(A)$ captures the non-forgotten atoms that remain in the filtered explanation.

Chapter 4

Xclingo

4.1 Introduction

We provide a tool called `xclingo`, which computes the support graphs for ASP programs. The tool further extends the ASP language with some so-called annotations that help the user design the produced explanations. Several examples of how to use these annotations are provided in Section 4.2.

`xclingo` relies on a meta-programming or reification method for computing the support graphs of a program. That is, the semantics of support graphs are specified in the `xclingo` ASP program that, together with a reification of the original program, whose answer sets correspond to the support graphs of a program. Both the reification's and the support graph's semantics specifications are explained and discussed in Section 4.3.

The internal architecture of the tool and some software design decisions are discussed in Section 4.4. This also includes additional advanced features such as the option to add *extensions* to `xclingo`. Extensions are snippets of ASP code that one can inject into the `xclingo` support graphs-computing program such that its behavior can be further extended in many ways.

4.2 Using Xclingo to Generate Explanations

4.2.1 Annotating a program to obtain Explanations

As an illustration, consider program P 4.1.

This program decides whether if a person is either innocent or sentenced to prison, depending on whether or not she has committed certain offenses. A person can be punished and therefore sentenced to prison if she drives after having ingested alcohol above the legal limit (namely 30 mg/l) or if she resisted authority (see lines 9 and 10). By default, a person is innocent (line 12) but can be sentenced to prison when punished (line 13). In particular, the program depicts 2 persons `gabriel` and `clare`. For `gabriel`, we imagine three scenarios: one where gabriel has an alcohol blood level of 40mg/l, one where `gabriel` resisted authority, and a third one where both situations happen. The events involved are represented by the atoms `alcohol(gabriel, 40)` and `resist(gabriel)`. The choice from line 7 generates the three scenarios. The `#show` from line 15 asks `clingo` to show the sentences for every person in the program.

```
 1  % dont_drive_drunk.lp
 2
 3  person(gabriel;clare).
 4  drive(gabriel). drive(clare).
 5  alcohol(clare, 5).
 6  % Either gabriel drove drunk or resisted to authority
 7  1{alcohol(gabriel, 40); resist(gabriel)}.
 8
 9  punish(P) :- drive(P), alcohol(P,A), A>30, person(P).
10  punish(P) :- resist(P), person(P).
11
12  sentence(P, innocent) :- person(P), not punish(P).
13  sentence(P, prison) :- punish(P).
14
15  #show sentence/2.
```

Program 4.1: Example *dont_drive_drunk.lp*.

A standard call to `clingo` by running

```
clingo 0 dont_drive_drunk.lp        (Command 4.1)
```

would produce the output in Output 4.1. There, we see that Program 4.1, has exactly 3 answer sets. Due to the `#show` sentence of line 15, we can only see `sentence/2` atoms in the solutions, thus one cannot distinguish the reason why gabriel was sentenced to prison in each scenario. To do so we would need

```
Answer: 1
sentence(clare,innocent) sentence(gabriel,prison)
Answer: 2
sentence(clare,innocent) sentence(gabriel,prison)
Answer: 3
sentence(clare,innocent) sentence(gabriel,prison)
SATISFIABLE
```

Output 4.1: Output for program P 4.1 after running Command 4.1.

to modify the original encoding, either to add extra `#show` statements, remove all of them, or to model the reason why somebody is punished, for example, as an extra argument to the `sentence` predicate.

Let us see now what kind of output we can obtain using `xclingo`. Essentially, `xclingo` works as a command line tool in the same way `clingo` does. When fed with files, the tool will interpret them as ASP programs extended with the `xclingo` markup annotation language and then will compute the explanations. To illustrate how this works, consider now program P 4.2, which extends program P 4.1 with some extra `xclingo` annotations. In this version, we see several annotations that include `trace`, `trace_rule` and `show_trace`. Each of them starts with %, so that `clingo` would interpret them as mere comments. As a result, running Command 4.1 over the annotated program P 4.2 would produce the exact same output we got in Output 4.1 for program P 4.1.

The `trace` annotation from line 12 associates a text describing the alcohol level with atoms of the form `alcohol(P, A)` for any `alcohol(P, A)` where `A` (the registered alcohol level) is greater than 30. The `trace_rule` annotations from lines 14, 17, 20 and 23, on the other hand, depend on the rule right below and will associate the corresponding text with any atom derived from such rules. Finally, similarly to

```
% annotated_dont_drive_drunk.lp

person(gabriel;clare).
drive(gabriel). drive(clare).
alcohol(clare, 5).


% Either gabriel drove drunk or resisted to authority
1{alcohol(gabriel, 40); resist(gabriel)}.


%!trace_rule {"% drove drunk (over 30mg/l)", P}.
punish(P) :- drive(P), alcohol(P,A), A>30, person(P).

%!trace_rule {"% resisted to authority", P}.
punish(P) :- resist(P), person(P).

%!trace_rule {"% is innocent by default",P}.
sentence(P, innocent) :- person(P), not punish(P).

%!trace_rule {"% has been sentenced to prison", P}.
sentence(P, prison) :- punish(P).

%!show_trace {sentence(P,S)} :- sentence(P,S).
#show sentence/2.
```

Program 4.2: Annotated *annotated_dont_drive_drunk*.

what `clingo` `#show` does, the `show_trace` annotation is used to select the subset of atoms that will be explained in the output. In this case, any occurrence of atom `sentence(P, S)`. In Appendix A, we will discuss the usage of each kind of annotation in full detail.

To use `xclingo` to obtain explanations for Program P 4.2, we would ran Commmand 4.2,

```
xclingo -n 0 0 annotated_dont_drive_drunk.lp
```
(Command 4.2)

obtaining the output in Output 4.2.

In contrast to what we saw in `clingo`'s output in Output 4.1, a user can now easily understand that the difference between the scenario in answer set 1 and answer set 2 is the reason why `gabriel` is being punished for. Besides, we see how answer set 3 has 2 explanations, following what was explained in Example 2 of Section 3.2 about some solutions potentially having several explanations. Also, it is straightforward to see how in answer set 3, *driving drunk* and *resisting to authority* are equally valid reasons to punish `gabriel`. In terms of structure, in `xclingo`'s output, we see the answer set section additionally divided into several *explanation* sections, each one featuring each computed support graph for the corresponding answer set. Inside each *answer set explanation*, we see a tree-like *atom explanation* for the atoms requested through `show_trace` annotations. The atom explanation start with a root node `*` from where it hangs the rest of the explanation. The nodes of the *tree* explanation are indented and connected through lines for easy reading. The parent nodes are in a lower level of indentation (i.e. located at the left) while child nodes are in the next indentation level. The parent-children relations in the tree must be understood as *cause-effect* relations where children nodes are the joint causes. Informally speaking, each indented connection can be read as a *because* or as a *is caused by*. For instance, the first atom explanation in Explanation 1.1, should be read as:

```
Answer: 1
##Explanation: 1.1
  *
  |__"gabriel has been sentenced to prison"
  |  |__"gabriel resisted to authority"

  *
  |__"clare is innocent by default"

##Total Explanations:   1
Answer: 2
##Explanation: 2.1
  *
  |__"gabriel has been sentenced to prison"
  |  |__"gabriel drove drunk (over 30mg/l)"
  |  |  |__"gabriel alcohol's level is 40"

  *
  |__"clare is innocent by default"

##Total Explanations:   1
Answer: 3
##Explanation: 3.1
  *
  |__"gabriel has been sentenced to prison"
  |  |__"gabriel drove drunk (over 30mg/l)"
  |  |  |__"gabriel alcohol's level is 40"

  *
  |__"clare is innocent by default"

##Explanation: 3.2
  *
  |__"gabriel has been sentenced to prison"
  |  |__"gabriel resisted to authority"

  *
  |__"clare is innocent by default"

##Total Explanations:   2
Models: 3
```

Output 4.2: Output for Program P 4.2 after running Command 4.2.

gabriel has been sentenced to prison *because* gabriel resisted authority.

Turn your attention now to the commands 4.1 and 4.2. In `clingo`'s command, we request the solver all the models (i.e. answer sets) by placing the option `0` right before the input files. Using any number $N > 0$ means requiring N answer sets at most. In the case of `xclingo`, in addition to handling the answer sets of the original program, it also gives the user the option to request an arbitrary number of explanations for each model. That is why the option `-n` in Command 4.2 is followed by two integers. The first is the number of answer sets of the original program that `xclingo` will consider at most, and the second is the number of maximum explanations `xclingo` will compute for each model. Note that this corresponds to the maximum number of support graphs considered, rather than the number of tree-like *atom explanations* we find inside each *answer set explanation*, this is determined by the atoms that match with the `show_trace` annotations inside the program. For example, Command 4.3 is asking for *all the explanations* only for the first answer set. Output 4.3 is the output we would obtain by running Command 4.3.

```
xclingo -n 1 0 annotated_dont_drive_drunk.lp
```
(Command 4.3)

```
Answer: 1
##Explanation: 1.1
  *
  |__"gabriel has been sentenced to prison"
  |  |__"gabriel resisted to authority"

  *
  |__"clare is innocent by default"

##Total Explanations:   1
Models: 1
```

Output 4.3: Output for Program 4.2 after running Command 4.3.

On the other hand, Command 4.4 is asking for only *one explanation* for each first three answer sets.

```
xclingo -n 3 1 annotated_dont_drive_drunk.lp
```
(Command 4.4)

In Output 4.4, which is the output we would obtain by running Command 4.4, we see how we only obtain one explanation in answer set 3 now.

Note the absence of cause for `clare` being innocent in all the answer sets. It is important to distinguish between the different types of questions that can be formulated and the different ways of answering them. Now, be aware that the kind of question that we are trying to explain here is "*How come* `clare` *to be innocent?*" and not "*Why is* `clare` *innocent instead of guilty*" or "*Why* `clare` *was not sentenced to prison*". For answering "*How come*" questions, we focus only on the positive part of the program as, in the real physical world, only events that happened can be actual causes for other events. Using events that did not happen to answer would be performing a counterfactual, which is appropriate for a different type of questions but not "*How come*" questions. For instance, in this case, the program poses that any person is innocent if there is no evidence of punishment. We could say that having `clare` been punished, would be a cause for being in prison, but not that the absence of punishment is a cause for being innocent. There is no need to mention all the crimes that `clare` could have

```
Answer: 1
##Explanation: 1.1
  *
  |__"gabriel has been sentenced to prison"
  |  |__"gabriel resisted to authority"

  *
  |__"clare is innocent by default"

##Total Explanations:   1
Answer: 2
##Explanation: 2.1
  *
  |__"gabriel has been sentenced to prison"
  |  |__"gabriel drove drunk (over 30mg/l)"
  |  |  |__"gabriel alcohol's level is 40"

  *
  |__"clare is innocent by default"

##Total Explanations:   1
Answer: 3
##Explanation: 3.1
  *
  |__"gabriel has been sentenced to prison"
  |  |__"gabriel drove drunk (over 30mg/l)"
  |  |  |__"gabriel alcohol's level is 40"

  *
  |__"clare is innocent by default"

##Total Explanations:   1
Models: 3
```

Output 4.4: Output for Program 4.2 after running Command 4.4.

committed but not, to explain how she has come to be innocent. Such an approach would involve updating the causes for `clare` being innocent any time new possible crimes were represented in the program (for instance drug trafficking, speeding while driving, etc.), even if the particular situation of `clare` would have not changed. However, if the posed question is "*Why* `clare` *was not sentenced to prison*", a valid answer would involve imagining events that have not occurred and which would have caused that, (this is, counterfactual reasoning), and giving the crimes that `clare` has not committed would be appropriate. In Section 5.5, we propose a way to answer this type of question with `xclingo` when they are explicitly posed by a user.

4.2.2 Tracing or Hiding Atoms

In Chapter 3, we have shown the definition of support graphs (see Definition 6) and how we can obtain explanations for any atom that match its corresponding derivation proofs (see Definition 8 and Example 3). However, by default, `xclingo` does not compute the explanations directly as the derivation proofs. For instance, in outputs 4.2, 4.3 and 4.4, we have seen that: (1) instead of the atom names, we obtain their corresponding text traces written by the user; and (2), any non-traced atom is *ommited* in the final explanation trees. Recall the *node forgetting* operation defined in Chapter 3 (Definition 10) as well and note how the tree explanations computed by `xclingo` are the proofs obtained from the support graph after forgetting any non-traced atom.

By default, `xclingo` considers any atom in the answer set as a *forgettable* atom and thus, it will be removed from the final explanations. To prevent this default behavior for a set of atoms, the user has to write *Trace* annotations. Each trace annotation defines a set of atoms that will be considered as *traced* (*not forgettable*) and therefore will be part of the tree explanations. In addition, it also allows the user to define parametrized text labels that replace the atoms when displaying the explanations. Inside the given text, the user can make use of several placeholders `%` to link the values of the variables referred to in the actual ground atoms. This link is done in the order the variables were placed in the annotation.

Another option to prevent `xclingo` to *forget* atoms is the `trace_rule` annotation. It works very similarly to `trace` annotations, except that they accompany particular rules explicitly and they only affect the atoms derived from this rule. To understand how this affects the explanations, please recall Definition 6 in Chapter 3. There we see that each atom in the graph has to be linked to a unique (labeled) rule ℓ. Unlike `trace` annotations, which affect any atom disregarding which rule they come from, `trace_rule` annotations only affect the atoms in the graph that were labeled with their particular rule's label.

As an example, take programs 4.3 and 4.4 where we use `trace_rule` and `trace` annotations respectively to trace the same atoms from the same ASP source code. As it can be easily seen, the `trace` annotations from Program 4.4 are intendedly written to trace the exact same set of atoms that the `trace_rule` annotations from Program 4.3 (i.e. the bodies of the trace annotations coincide with the corresponding bodies of the two rules dor deriving `punish(P)`). The set of atoms that are being *traced* are the same. Both programs have the same unique answer set A.

$$\begin{aligned} A = \{\{&resist(gabriel), alcohol(gabriel, 40),\\ &drive(gabriel), punish(gabriel)\}\} \end{aligned}$$

Besides, this answer set has exactly the two support graphs that are depicted in Figure 4.1. The labels

```
% only_tracerules.lp

resist(gabriel).
alcohol(gabriel,40).
drive(gabriel).

%!trace_rule {"% resisted authority",P}.
punish(P) :- resist(P).

%!trace_rule {"% drove drunk", P}.
punish(P) :- drive(P), alcohol(P, A), A > 30.

%!show_trace {punish(gabriel)}.
```

Program 4.3: Program *dont_drive_drunk* using only `trace_rule`.

```
% only_traces.lp

resist(gabriel).
alcohol(gabriel,40).
drive(gabriel).

punish(P) :- resist(P).
punish(P) :- drive(P), alcohol(P, A), A > 30.

%!trace {punish(P), "% drove drunk", P} :- drive(P), alcohol(P, A), A > 30.
%!trace {punish(P), "% resisted authority",P} :- resist(P).

%!show_trace {punish(gabriel)}.
```

Program 4.4: Program *dont_drive_drunk* using only `trace`.

$$\begin{tikzcd}
\ell_1 : resist(gabriel) \arrow[d] & \ell_2 : alcohol(gabriel, 40) \arrow[dr] & \ell_3 : drive(gabriel) \arrow[d] \\
\ell_4 : punish(gabriel) & & \ell_5 : punish(gabriel)
\end{tikzcd}$$

Figure 4.1: Support graphs for the only answer set from programs 4.3 and 4.4.

used in the graphs are taken from the rules in the order they appear in the programs (i.e. the first rule for `punish(P)` takes label ℓ_4 and the second takes label ℓ_5). Note how the causes for the atom `punish(gabriel)` change depending on the rule being used to label it. We would obtain these two graphs regardless of which of both programs we would use to explain the answer set. However, the explanations got by `xclingo` do not look the same. If we run Command 4.5 with Program 4.3 we obtain the explanations in Output 4.5

```
xclingo -n 0 0 only_tracerules.lp -auto-tracing=facts
```
(Command 4.5)

```
Answer: 1
##Explanation: 1.1
  *
  |__"gabriel resisted authority"
  |  |__resist(gabriel)

##Explanation: 1.2
  *
  |__"gabriel drove drunk"
  |  |__drive(gabriel)
  |  |__alcohol(gabriel,40)

##Total Explanations:   2
Models: 1
```

Output 4.5: Explanations obtained for Program 4.3 after running Command 4.5.

In this output, we can see how each text label is displayed only when their corresponding rule is being used to explain the atom. This is thanks to the `trace_rule` annotation special behavior. However, in Output 4.6 we can see the different output we get using only `trace` annotations. As you can see, both

```
Answer: 1
##Explanation: 1.1
    *
    |__"gabriel drove drunk";"gabriel resisted authority"
    |  |__resist(gabriel)

##Explanation: 1.2
    *
    |__"gabriel drove drunk";"gabriel resisted authority"
    |  |__drive(gabriel)
    |  |__alcohol(gabriel,40)

##Total Explanations:   2
Models: 1
```

Output 4.6: Explanations obtained after running Command 4.5 for Program 4.4 instead.

text labels, being drunk and resisting, are being displayed in both explanations `1.1` and `1.2`, disregarding which rule is being used, even when in the case of explanation `1.2` the cause of the punishment was not resisting.

This example tries to illustrate the contrast between `trace_rule` and `trace` annotations. It is important to understand this difference to correctly design the desired explanations. Recalling Definion 6 of support graphs, only one rule from the program can be used in a graph to justify an atom. When, given an answer set, we have several rules for justifying an atom, we produce several graphs in which we include only one of the alternative rules. The `trace_rule` annotations are bonded to the rule and thus, their text only affects to graphs in which the corresponding rule was selected. In contrast, `trace` annotations are bonded to atoms and therefore their text affects all possible graphs. In the previous example, we have two rules that can justify `punish(gabriel)`. In Outputs 4.6 and 4.5, explanations `1.1` and `1.2` respectively correspond to the support graph for which each rule was selected instead of the other. However, note how `trace` annotations texts appear in both explanations. In terms of support graphs, we can think on `trace` annotations as if we were introducing a new rule $\ell : p' \leftarrow p \wedge C$ where p is the atom we are tracing, ℓ is the text we are associating to atom p, C is the condition we introduce as part of the `trace` annotation (i.e. the body of the annotation), and p' is an auxiliary atom. As this fresh rule is the only one supporting atom p', it has to be used to label the atom in any support graph. The appearances of p in the rest of the program would be then replaced by p' to keep the correct causal chains between the atoms.

Let us end this section with a note on how to organize the code regarding annotations. In general, keeping the traces independent from the code using `trace` annotations is always a better option. It enhances the cleanliness and reusability of the source code. Indeed, several different annotation files can be written with different purposes such as generating explanations for different users or in different languages. However, as we have just seen, for obtaining particular explanations, sometimes we need to use `trace_rule` annotations that must be placed next to their rules and, unfortunately, are impossible to separate in independent modules rules for now.

4.2.3 Obtaining Explanations without Manually Tracing Atoms

Another possibility allowed by `xclingo` to obtain explanations without adding any `trace` or `trace_rule` annotation using the command line option `-auto-tracing`. When this option is used with the value `-all`, `xclingo` will add an additional trace to *every* atom in the answer set. The text of this automatically added trace will match the atom itself (i.e. an automatic trace for the atom `sentence(gabriel, prison)` will be the text *sentence(gabriel, prison)*). Leveraging from this option, one could obtain explanations for non-annotated Program P 4.1, but we would still have to add at least one `show_trace` annotation or `xclingo` would not compute an explanation for any atom. For instance, if we take that program and an additional file containing (i.e. *showtrace.lp*) only the annotation

```
%!show_trace sentence(P,S) :- sentence(P,S).
```

and we run `xclingo` with the option `-auto-tracing=all` like in Command 4.6

```
xclingo -n 0 0 dont_drive_drunk.lp showtrace.lp -auto-tracing=all
```
(Command 4.6)

we would obtain the output in Output 4.7. In this output, we see how all the (ground) atoms that intervened in the derivation of the explained atom are shown in the explanation. Since all the atoms in the answer set are traced, the explanation trees shown match the derivation proofs for the explained atoms. The automatic tracing option also admits the `facts` value, which will produce automatic traces only for the facts of the program (i.e. those derived from rules with an empty body).

Of course, the `-auto-tracing` option can be mixed with custom `trace` and `trace_rule` annotations. For instance, consider now the execution of Command 4.7 below:

```
Answer: 1
##Explanation: 1.1
  *
  |__sentence(gabriel,prison)
  |  |__punish(gabriel)
  |  |  |__resist(gabriel)
  |  |  |__person(gabriel)

  *
  |__sentence(clare,innocent)
  |  |__person(clare)

##Total Explanations:   1
Answer: 2
##Explanation: 2.1
  *
  |__sentence(gabriel,prison)
  |  |__punish(gabriel)
  |  |  |__drive(gabriel)
  |  |  |__alcohol(gabriel,40)
  |  |  |__person(gabriel)

  *
  |__sentence(clare,innocent)
  |  |__person(clare)

##Total Explanations:   1
Answer: 3
##Explanation: 3.1
  *
  |__sentence(gabriel,prison)
  |  |__punish(gabriel)
  |  |  |__drive(gabriel)
  |  |  |__alcohol(gabriel,40)
  |  |  |__person(gabriel)

  *
  |__sentence(clare,innocent)
  |  |__person(clare)

##Explanation: 3.2
  *
  |__sentence(gabriel,prison)
  |  |__punish(gabriel)
  |  |  |__person(gabriel)
  |  |  |__resist(gabriel)

  *
  |__sentence(clare,innocent)
  |  |__person(clare)

##Total Explanations:   2
Models: 3
```

Output 4.7: Output for Program P 4.1 after running Command 4.6.

```
xclingo -n 0 0 dont_drive_drunk.lp showtrace.lp -auto-tracing=facts
```
(Command 4.7)

which use the `-auto-tracing=facts` option instead, where `showtrace.lp` now contains the following two annotations:

```
% showtrace.lp

%!show_trace {sentence(P,S)} :- sentence(P,S).
%!trace {sentence(P,S), "Sentece for %: %",P, S} :- sentence(P,S).
```

This call produces Output 4.8

Furthermore, if an atom has more than one associated text trace (for instance, the automatically generated one and a user-defined one), the tree explanation will show all of them. For example, if we just change the option `facts` by `all` in Command 4.7, any `sentence/2` atom will have two text traces: the automatic one and the custom one (inside `traces.lp`). As a result, we would obtain explanations like the following one,

```
    *
    |__"Sentece for gabriel: prison";sentence(gabriel,prison)
    |  |__punish(gabriel)
    |  |  |__drive(gabriel)
    |  |  |__alcohol(gabriel,40)
    |  |  |__person(gabriel)
```

Where both traces for `sentence(gabriel, prison)` are shown separated by a semicolon. This can be especially useful while designing your custom `trace` and `trace_rule` annotations, to check if they are being applied to the atoms as expected.

4.2.4 Muting ASP Code to Avoid Causal Links

When explaining ASP programs, `xclingo` assumes every rule in the input program depicts a cause-effect relation, where the atoms in the head are the effects and the literals in the body are the *actual causes*. However, as soon as we leave literature examples, this becomes a false assumption for most real ASP applications. Even in very causality-oriented applications such as diagnosis, we can often find non-causal rules among standard ASP implementations. A very typical example where the causal assumption breaks is when an ASP rule is used to complete the domain of a predicate. For instance the rule:

```
person(S) :- student(S).
```

means that *students are also persons* and is used to complete the person list without repeating all the (already declared) student names. However, it does not mean that *S is a person because she is a student* or that *being a student cause an entity to be a person*. `xclingo` will assume this latter meaning by default, though. We can prevent this assumption by telling `xclingo` which rules are not meant to model causal-effect relations. To this aim, the users can make use of *mute* (i.e. `mute`) and *mute body* (i.e. `mute_body`) annotations. They respectively work in an analogous way to `trace` and `trace_rule` (see Section 4.2.2). A `mute` annotation defines a set of atoms and marks them as *muted atoms* meaning that any will not be considered as a cause for any other atom. On the other hand, a `mute_body` annotation marks a rule as *muted rule* removing the cause-effect relations between the head and body of that particular rule. In the same way `trace` and `trace_rule` indirectly define the set of atoms over which `xclingo` applies the

```
Answer: 1
##Explanation: 1.1
  *
  |__"Sentece for gabriel: prison"
  |  |__resist(gabriel)
  |  |__person(gabriel)

  *
  |__"Sentece for clare: innocent"
  |  |__person(clare)

##Total Explanations:   1
Answer: 2
##Explanation: 2.1
  *
  |__"Sentece for gabriel: prison"
  |  |__drive(gabriel)
  |  |__alcohol(gabriel,40)
  |  |__person(gabriel)

  *
  |__"Sentece for clare: innocent"
  |  |__person(clare)

##Total Explanations:   1
Answer: 3
##Explanation: 3.1
  *
  |__"Sentece for gabriel: prison"
  |  |__drive(gabriel)
  |  |__alcohol(gabriel,40)
  |  |__person(gabriel)

  *
  |__"Sentece for clare: innocent"
  |  |__person(clare)

##Explanation: 3.2
  *
  |__"Sentece for gabriel: prison"
  |  |__person(gabriel)
  |  |__resist(gabriel)

  *
  |__"Sentece for clare: innocent"
  |  |__person(clare)

##Total Explanations:   2
Models: 3
```

Output 4.8: Output for Program P 4.1 after running Command 4.7.

```
 1  % brangelina.lp
 2
 3  wed(angelina,billy,2000).   divorced(angelina,billy,2003).
 4  wed(brad,jennifer,2000).    divorced(brad,jennifer,2005).
 5  wed(brad,angelina,2014).    divorced(brad,angelina,2019).
 6  wed(billy,connie,2014).
 7
 8  person(A1) :- wed(A1,A2,Y).
 9  person(A2) :- wed(A1,A2,Y).
10  %!mute {person(P)}.
11
12  %!mute_body.
13  wed(A,B,Y) :- wed(B,A,Y).
14  %!mute_body.
15  divorced(A,B,Y) :- divorced(B,A,Y).
16
17  unwed(A1,A2,YM) :- wed(A1,A2,YM), divorced(A1,A2,YD), YM<YD.
18
19  married(A1) :- wed(A1,A2,YM), not unwed(A1,A2,YM).
20  single(P) :- person(P), not married(P).
21
22  %!show_trace {single(P)}.
23  %!show_trace {married(P)}.
```

Program 4.5: Example *brangelina.lp*

forgetting operation discussed in Section 3.3, `mute` and mute rule define the sets over which the edge pruning operation is performed. Recalling Definition 9 of edge pruning, any outgoing edge from a muted atom is pruned, and any incoming edge on an atom labeled with a muted rule's label is pruned.

To illustrate the behavior of these annotations and their utility consider Program P4.5, The program has several facts about weddings, divorces, and their associated years. Predicate `person(P)` just collects anybody mentioned in any wedding. Besides, predicates `wed(A1,A2,Y)` and `divorced(A1,A2,Y)` are symmetric in their arguments `X` and `Y`. Predicate `unwed(A1,A2,Y)` points out a wedding `wed(A1,A2,Y)` that became ineffective by a later divorce. Finally, predicates `married(A1)` and `single(P)` indicate the current marital status of some person `X`. Note that the rule for `single(P)` is formulated as a *default*: somebody is single if we cannot prove she is currently married.

Note that one could be also interested in showing the complete list of marriages and divorces to prove that someone is single. This would be modeled in a different way. Here, the default perspective adopts the idea that someone is single if not proven to be currently married and the previous history is not disclosed: it may even be subject to privacy and data protection constraints.

Program 4.5 has a unique answer set for which `xclingo` generates the unique explanation shown in Output 4.9 by using the Command 4.8.

```
xclingo -n 0 0 brangelina.lp -auto-tracing=all
```
(Command 4.8)

Note the use of `mute` and `mute_body` annotations in lines 10, 12 and 14. Rules from lines 13 and 15 only model the symmetric properties of predicates `wed(A1,A2,Y)` and `divorced(A1,A2,Y)`, but they are not meant to model any causal relation. The `mute_body` annotations from lines 12 and 13 are preventing those rules from propagating cause; i.e. is `xclingo` will not use the rules to create edges in the support graphs. The `mute` annotation of line 10 is making any `person(P)` atom as *muted*, meaning that it cannot be a cause for another atom. If not *muted*, the `person` atoms would take part in explaining `single`

```
Answer: 1
##Explanation: 1.1
  *
  |__single(angelina)

  *
  |__single(brad)

  *
  |__single(jennifer)

  *
  |__married(billy)
  |  |__wed(billy,connie,2014)

  *
  |__married(connie)
  |  |__wed(connie,billy,2014)
  |  |  |__wed(billy,connie,2014)

##Total Explanations:   1
```

Output 4.9: Output for Program 4.5 after running Command 4.8.

atoms (see rules in line 20), as `xclingo` would conclude that a being married is a reason to be a person. Moreover, if a person gets married more than once, then it has several explanations for being a person. Output 4.10 corresponds to a program in which the `person/1` atoms were not muted.

In fact, this Output shows only a part of the complete output: we are only explaining `single(P)` atoms for brevity. As we can see, not only `person(P)` appears now as a cause of `single(P)`, but `person(P)` is caused by the weddings as well. Also, as we anticipated, we have more than one explanation now due to each wedding being an alternative cause for `person`. Moreover, `wed(A1,A2,Y)` is a symmetric predicate, meaning that for each wedding a person `P` was involved in, we are providing two causes for `person(P)`: `wed(P,A,Y)` and `wed(A,P,Y)`. Since `brad` and `angelina` had a total of two weddings, and `jennifer` had one, this leads to a total of 32 explanations.

4.2.5 Explanation Explosion

As demonstrated in the example of the previous section, *muting* atoms and rules is not only a way to customize the explanations, but sometimes a needed tool to design proper explanations. Not *muting* non-causal rules can also lead to an undesired explosion in the number of explanations.

Consider Program P 4.6. The program is a variation (introduced by Fandinno in [49]) of the classical *Firing Squad Scenario* introduced by J.Pearl in [81] for causal counterfactuals (although we do not use it for that purpose here). We have an army distributed in $maxn >= 1$ squads of three soldiers each, a captain and two riflemen (`a` and `b`) for each squad. We place the squads on a sequence of $maxn$ consecutive hills. An unfortunate prisoner is at the last hill $maxn - 1$, and is being aimed at by the last two riflemen. At each hill i, the two riflemen a_i and b_i will fire (respectively represented by the atoms `fire_a(i)` and `fire_b(i)`) if their captain gives a signal to fire represented by atom `signal(i)`. But then, the captain at the net hill will give a signal to fire (this is `signal(i+1)`) if she hears a shot from the previous hill i in the distance. Suppose the captain at hill 0 gives a signal to fire.

```
##Explanation: 1.1
  *
  |__single(angelina)
  |  |__person(angelina)
  |  |  |__wed(angelina,brad,2014)

  *
  |__single(brad)
  |  |__person(brad)
  |  |  |__wed(brad,angelina,2014)

  *
  |__single(jennifer)
  |  |__person(jennifer)
  |  |  |__wed(brad,jennifer,2000)

##Explanation: 1.2
  *
  |__single(angelina)
  |  |__person(angelina)
  |  |  |__wed(angelina,brad,2014)

             ( . . . )
```

Output 4.10: Output for Program 4.5 after running Command 4.8, when unmuting atom `person(P)`.

```
% chain_of_firing_squads.lp

#const maxn=7.
signal(0).

fire_a(N) :- signal(N),N<maxn.
fire_b(N) :- signal(N), N<maxn.

signal(N+1) :- fire_a(N), N<maxn.
signal(N+1) :- fire_b(N), N<maxn.

%!show_trace {signal(maxn)}.
```

Program 4.6: Example *chain_of_firing_squads.lp*.

This program is positive and has only one answer set (the least model). However, this answer set has a total of 2^{maxn} explanations since any atom `signal(i+1)` for any hill i can be explained by `fire_a(i)` or `fire_b(i)`. Output 4.11 shows part of `xclingo`'s output after Command 4.9.

```
xclingo -n 0 0 chain_of_firing_squads.lp -auto-tracing=all
```
(Command 4.9)

```
##Explanation: 1.128
  *
  |__signal(7)
  |  |__fire_b(6)
  |  |  |__signal(6)
  |  |  |  |__fire_a(5)
  |  |  |  |  |__signal(5)
  |  |  |  |  |  |__fire_b(4)
  |  |  |  |  |  |  |__signal(4)
  |  |  |  |  |  |  |  |__fire_a(3)
  |  |  |  |  |  |  |  |  |__signal(3)
  |  |  |  |  |  |  |  |  |  |__fire_a(2)
  |  |  |  |  |  |  |  |  |  |  |__signal(2)
  |  |  |  |  |  |  |  |  |  |  |  |__fire_b(1)
  |  |  |  |  |  |  |  |  |  |  |  |  |__signal(1)
  |  |  |  |  |  |  |  |  |  |  |  |  |  |__fire_a(0)
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__signal(0)

##Total Explanations:   128
```

Output 4.11: Extract of the output for Program P4.6 after running Command 4.9.

Note the impact that this observation has on particular real-world scenarios. For instance, this of *explanation explosion* may also easily happen in temporal scenarios where some events are caused by events in the previous timesteps. Any system trying to obtain all explanations will face an exponential computation cost, even for positive programs (as it is the case of the previous example). The most straightforward solution to this problem is obtaining just one (or a custom limited number of) explanation. This can be done easily with `xclingo`, as we have already shown. A perhaps more convenient way to tackle the problem is by selecting among the explanations by defining criteria. In `xclingo`, this can use a feature called extensions. In Section 4.4.3, we explain how to create and apply extensions.

4.2.6 Explaining aggregates

`xclingo` also provides explanations for aggregates using a first straightforward approach (we discuss its limitations later in this section and in Section 4.3.2). In short words, any atom contributing to the final value computed by the aggregate is considered as a joint cause, among the rest of the atoms supporting the rule.

To demonstrate how this works consider Program 4.7.

In this program, we count how many objects are held by `mary` at different time steps. The predicate `held_by(O,E,T)` represents that the object `O` is being held by the entity `E` at time step `T`, whereas the predicate `numberObjectsbyEntityatTime(N,E,T)` indicates the number of objects `N` being held by `E` at `T`. The rule in line 11 uses a `#count` aggregate counting the different objects `O` being held by `E` at `T`, given their corresponding `held_by` atoms. A couple of `mute_body` annotations prevent domain predicates

```
held_by(football,mary,0).
held_by(apple,mary,0).
held_by(football,mary,1).

time(T):-held_by(O,E,T).
entity(E):-held_by(O,E,T).

numberObjectsbyEntityatTime(N,E,T):- N=#count{O: held_by(O,E,T)}, entity(E),time(T).

%!show_trace {numberObjectsbyEntityatTime(N,mary,T)}.
```

Program 4.7: Example *holding_objects.lp*.

`entity/1` and `time/1` from propagating cause. A `show_trace` annotation finally requests explanations for the number of objects that `mary` holds at the different time steps.

After running Command 4.10, `xclingo` obtains Output 4.12.

```
xclingo -n 0 0 polluted_river.lp -auto-tracing=all
```
(Command 4.10)

```
Answer: 1
##Explanation: 1.1
    *
    |__numberObjectsbyEntityatTime(2,mary,0)
    |  |__entity(mary)
    |  |__time(0)
    |  |__held_by(football,mary,0)
    |  |__held_by(apple,mary,0)

    *
    |__numberObjectsbyEntityatTime(1,mary,1)
    |  |__entity(mary)
    |  |__time(1)
    |  |__held_by(football,mary,1)

##Total Explanations:   1
Models: 1
```

Output 4.12: Output for Program 4.7 after running Command 4.10.

As it can be seen, the `held_by` atoms are considered a cause of the `numberObjectsbyEntityatTime`, at the same level as the other atoms `entity` and `time`. In other words, `xclingo` considers as equally contributive causes all the atoms involved in the set of values to be aggregated and the rest of the body of the rule. A more elaborated approach would be, to reason about the necessary and/or the sufficient causes that make the rule true. This kind of analysis seems more useful in explaining monotone aggregates. For instance, consider Program 4.8. It consists of 3 entities factory `a`, factory `b` and farm `c`, each of them has contributed to polluting river `rhin` with a particular amount of chemical pollution (30, 30 and 60, respectively) and an additional amount of 10 nutrient pollution in the case of farm `c`. A river is considered polluted if the total amount of pollution is greater than 50. This is represented in the rule from line 13, which uses a `#sum` aggregate in its body. Finally, a `show_trace` annotation asks why river `rhin` is polluted. After running Command 4.11, `xclingo` obtains Output 4.13.

```
xclingo -n 0 0 polluted_river.lp -auto-tracing=all
```
(Command 4.11)

```
% polluted_river.lp

river(rhin).
factory(a;b).
farm(c).

chemical(a, rhin, 30). chemical(b, rhin, 60). chemical(c, rhin, 10).
nutrient(c, rhin, 30).

entity(E) :- factory(E).
entity(E) :- farm(E).

polluted_river(River) :-
    river(River),
    #sum{P1: entity(E), chemical(E, River, P1); P2: entity(E), nutrient(E, rhin, P2)}=Sum,
    Sum>50.

%!show_trace {polluted_river(rhin)}.
```

Program 4.8: Example *polluted_river.lp*.

```
##Explanation: 1.1
  *
  |__polluted_river(rhin)
  |  |__river(rhin)
  |  |__entity(a)
  |  |  |__factory(a)
  |  |__entity(b)
  |  |  |__factory(b)
  |  |__entity(c)
  |  |  |__farm(c)
  |  |__chemical(a,rhin,30)
  |  |__chemical(b,rhin,60)
  |  |__chemical(c,rhin,10)
  |  |__nutrient(c,rhin,30)

##Total Explanations:   1
```

Output 4.13: Output for Program 4.8 after running Command 4.11.

As can be seen, any contribution from any company acts as a joint cause. Moreover, note that all the ground possibilities to derive both conditional atoms appear in the explanation. However, in this case, `b` alone is sufficient to consider the river as polluted, as also are `a` together with `c`. Although this kind of analysis is not possible natively in `xclingo`, it is possible via the use of extensions, a feature which is discussed in Section 4.4.3. However, is not clear if providing a subset of atoms would be the default behavior, or if it should be the user the one that should select how the aggregate should be explained in each case. This topic still requires more research.

Finally, as the previous explanations contain so much detail, we provide an annotated version in Program 4.9 that includes some `trace`, `trace_rule` and `mute` annotations. Additionally, it modifies the predicate `entity` to include the type of entity (`factory` or `farm`), for using it as part of a text trace. When running Command 4.12, `xclingo` obtains the explanations found in Output 4.14.

```
xclingo -n 0 0 polluted_river_annotated.lp
```
(Command 4.12)

```
% poluted_river_annotated.lp

river(rhin).
factory(a;b).
farm(c).

chemical(a, rhin, 30). chemical(b, rhin, 60). chemical(c, rhin, 10).
nutrient(c, rhin, 30).

entity(E, factory) :- factory(E).
entity(E, farm) :- farm(E).
%!mute {entity(E,T)}.

%!trace {chemical(E,R,P), "% % contributed % to chemical pollution", T, E, P} :- chemical(E,R,P), entity(E, T).
%!trace {nutrient(E,R,P), "% % contributed % to nutrient pollution", T, E, P} :- nutrient(E,R,P), entity(E, T).

%!trace_rule {"River % is considered polluted (total pollution %)", River, Sum}.
polluted_river(River) :-
    river(River),
    #sum{P1: entity(E, T), chemical(E, River, P1); P2: entity(E, T), nutrient(E, rhin, P2)}=Sum,
    Sum>50.

%!show_trace {polluted_river(rhin)}.
```

Program 4.9: An annotated version of *polluted_river.lp*.

```
Answer: 1
##Explanation: 1.1
  *
  |__"River rhin is considered polluted (total pollution 100)"
  |  |__"company a contributed 30 to chemical pollution"
  |  |__"company b contributed 60 to chemical pollution"
  |  |__"farm c contributed 10 to chemical pollution"
  |  |__"farm c contributed 30 to nutrient pollution"

##Total Explanations:   1
```

Output 4.14: Output for Program 4.9 after running Command 4.12.

4.2.7 Explaining unsatisfiable programs

Up to this point, we have shown how to use `xclingo` features to design and obtain explanations for the answer sets of ASP programs. In all cases, both the tool and the method explained in Chapter 3 need an actual answer set to start computing explanations. When the program P has no answer set, $SM(P) = \emptyset$, there are no support graphs to be shown, and the user would get no explanation at all. Yet, `xclingo` implements a simple method to avoid this situation, pointing at the constraints that contribute in a higher degree to the unsatisfiability of P.

More precisely, when facing an unsatisfiable program, `xclingo` can point at which constraints are being "active" and explain why is that. By "active" we mean that if we removed the set of constraints C to get program $P' = P \setminus C$, we would get at least some model $M \in SM(P')$ such that $M \models B$ for all constraint $(\bot \leftarrow B) \in C$ To enable this feature, the user can decide which constraints can be used in this kind of explanation by adding `trace_rule` annotations right before them, as with regular rules.

After finding out that the original program has no answer set, `xclingo` will *relax* (i.e. create an auxiliary head for the constraint rule, disabling it) the traced constraints in the original program and try to solve it again.

If the program is still unsatisfiable, then `xclingo` will answer `UNSAT`. But if this scall obtains some answer set, then `xclingo` will explain (the head atoms of) these relaxed constraints as if they were positive rules.

Of course, relaxing a constraint will cause the program to generate incorrect solutions when solving. Within these solutions, we will find at least one but potentially of these auxiliary atoms indicating that some relaxed constraints are being fired. Since this option is mainly oriented toward debugging programs expected to be satisfiable, we assume that the user will be interested in cases where relaxed constraints are fired as less as possible. In other words: *the closest to a correct solution*. To this aim, `xclingo` will minimize the number of these auxiliary atoms to find such a model.

To illustrate how this works, consider Program 4.10. The program includes a graph with 5 vertices and some edges and tries to find Hamiltonian paths on it. Predicate `in/2` represents the edges included in the path and the predicate `reached/1` gives the vertices that the path passes through. The first two constraints ensure each vertex is only crossed once, and the last constraint ensures all vertices are included in the path. Note how all constraints are *traced* with customized texts.

Since any of the included edges impinges in vertex 5, it cannot be a Hamiltonian path in the graph. However, by running Command 4.13, we can obtain some hints about why the program has no model.

```
xclingo -n 0 0 hamiltonian_path.lp
```
(Command 4.13)

Output 4.15 shows `xclingo` output. First, we can see how the first attempt to obtain models for Program 4.10 ends in an unsatisfiable result. After that, `xclingo` relaxes the traced constraints and starts minimizing the number of times they are fired. First models are not so helpful, since a lot of constraints are being fired. For instance, the first obtained model corresponds to an empty path, where any vertex is reached. The last one is more interesting though, where we see that only vertex 5 is unreachable and at this point, the user probably realizes that there is a mistake in the input data (for instance, vertex 5 should not exist, perhaps there is some missing edge).

```
UNSATISFIABLE
Relaxing constraints... (mode=minimize)
Answer: 1
##Explanation: 1.1
  *
  |__"I can't reach vertex 1 from vertex 1"

  *
  |__"I can't reach vertex 2 from vertex 1"

  *
  |__"I can't reach vertex 3 from vertex 1"

  *
  |__"I can't reach vertex 4 from vertex 1"

  *
  |__"I can't reach vertex 5 from vertex 1"

##Total Explanations:   1
Answer: 2
##Explanation: 2.1
  *
  |__"I can't reach vertex 1 from vertex 1"

  *
  |__"I can't reach vertex 2 from vertex 1"

  *
  |__"I can't reach vertex 5 from vertex 1"

##Total Explanations:   1
Answer: 3
##Explanation: 3.1
  *
  |__"I can't reach vertex 5 from vertex 1"

##Total Explanations:   1
Models: 3
```

Output 4.15: Output for Program P 4.10 after running Command 4.13.

```
% hamiltonian_path.lp

vtx(1..5).
edge(1,2).
edge(2,3). edge(2,4).
edge(3,1). edge(3,4).
edge(4,3). edge(4,1).

{in(X,Y)} :- edge(X,Y).

%!trace_rule {"I can't pick outgoing edges %->% and %->%",X,Y,X,Z}.
:- in(X,Y), in(X,Z), Y!=Z.

%!trace_rule {"I can't pick incoming edges %->% and %->%",X,Z,Y,Z}.
:- in(X,Z), in(Y,Z), X!=Y.

reached(V) :- in(1, X).
reached(Y) :- reached(X), in(X,Y).

%!trace_rule {"I can't reach vertex % from vertex 1",X}.
:- vtx(X), not reached(X).

#show in/2.
```

Program 4.10: Example *hamiltonian_path.lp*.

Note how, when explaining unsatisfiable programs, `xclingo` automatically shows the explanations of the traced constraints without the need for any `show_trace` annotation. Whereas user-written `show_trace` annotations will be ignored.

Although this will be more widely discussed in Chapter 5, let us advance how this feature does not cover all the types of causal queries related to explaining UNSAT programs. It is especially useful, however, when debugging programs that *should be satisfiable*. In that sense, we could consider the explanations as *error prompts*. For that purpose, tracing all the constraints should not harm, as they will only be used when the program *fails* for unknown reasons (i.e. it gives no answer set). Although it would be convenient to accompany them with a good set of annotations for the atoms involved in the constraints to obtain informative error messages.

4.3 ASP Implementation

For computing the support graphs of ASP programs, `xclingo` leverages meta-programming. This is because the tool uses an ASP program for computing the support graphs of another (input) ASP program. Due to the declarative nature of ASP, this provides the implementation with enhanced maintainability.

In Section 3.4 of Chapter 3, we provide an encoding for computing support graphs which was proved to be consistent with respect to the definitions. The encoding used by `xclingo` that will be discussed in this section, although equivalent, extends it with some additional features, as for instance, the annotations. During the discussion, we will refer to the original encoding where appropriate.

Of course, since the code computing the explanations is ASP code, the input of such process must also be ASP code. This input includes the models of the original program itself as well as its models. Figure 4.2 shows how the different ASP components of `xclingo` interact to obtain the explanations. In

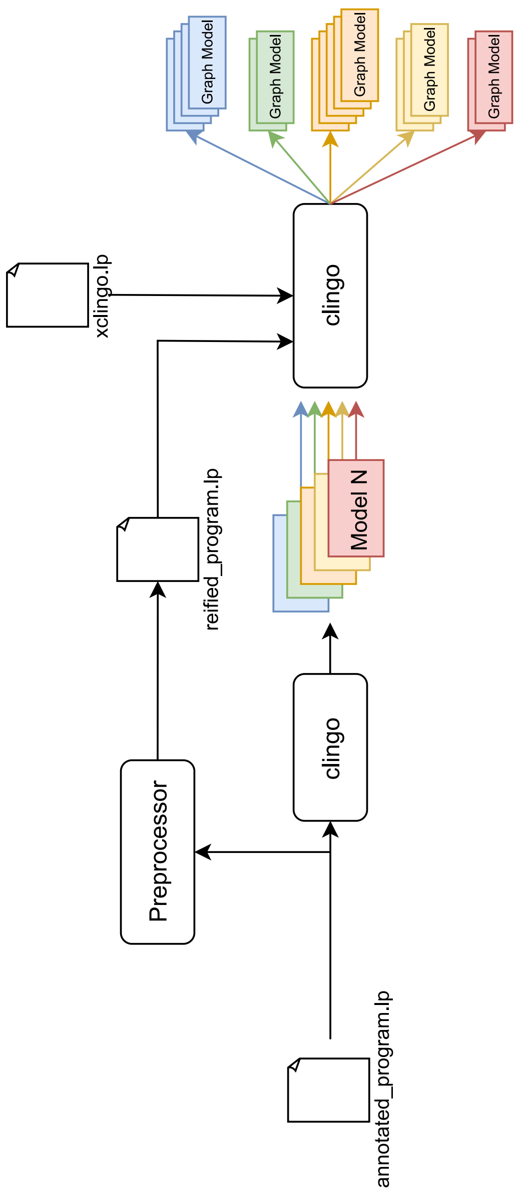


Figure 4.2: ASP components of `xclingo`. The diagram shows the flow from the original program to the computed support graphs..

the case of the original program (in the figure called *annotated_program.lp*), it is firstly translated into a reified version (in the figure called *reified_program.lp*). This reification follows the implementation explained in Section 3.4 but additionally includes the annotations. The models of the original program are obtained by using `clingo` (normally) to solve the program. Finally, the reification and the model (one at a time) are put together with the core `xclingo` ASP program (in the figure called *xclingo.lp*). The program resulting from the union of these three will be called *explainer program*. When solved, each answer set of the explainer program will be one of the explanations of the corresponding model and so we refer to them in the figure as *Graph Model*. In this way, `xclingo` also adopts the unusual problem-solving methodology in ASP with a one-to-one correspondence between the answer sets of the encoding and the solutions to a problem, in this case, the explanations of another answer set.

Understanding the ASP specification of the explainer program is perhaps not so important for the final users, whose only interest is to obtain explanations that meet their requirements. It can be so, however, for the ASP engineer who has to write annotated programs that fulfill such requirements. As we have seen, the `xclingo` markup annotation language provides significant flexibility for designing the final explanations. But understanding the semantics of the explainer program, coupled with the fact that this is a declarative specification, should allow advanced ASP engineers to *extend* such program to achieve more ambitious functionalities. Once the expert understands the internal representation of the explanations graphs, that could be exploited for finding out new relations within the graph like the *indirect causes* of an atom or writing *complex minimization* criteria regarding the explanations, among many other ideas. Indeed, some of the most used `xclingo` features like the auto-tracing are implemented as a *one ASP-line* extension of the explainer program. That extension and others included by default in `xclingo` will be discussed in Section 4.4.3, as well as how to proceed to write custom extensions.

For the rest of this section, we will discuss the building and the semantics of `xclingo`'s explainer program. More precisely, the reification generated for the original program as well as the `xclingo` core ASP code to compute the explanations.

4.3.1 Representation of the explained model

The first important predicate of the explainer program is `_xclingo_model/1`, where its first (and only) argument is meant to be an atom. Its meaning is that the atom is true for the corresponding model. This predicate corresponds to the $as/1$ predicate defined in the original encoding (see Section 3.4 of Chapter 3). Since the solving of the original program computes the models, the atoms within them are collected by `xclingo`. The process to include them in the explainer program just involves wrapping the atoms within the predicate.

For instance, if we have an answer set consisting of the following atoms,

$$\{alcohol(clare, 5), person(gabriel), person(clare), drive(gabriel)\}$$

It would be translated into

```
_xclingo_model(alcohol(clare,5)).
_xclingo_model(person(gabriel)).
_xclingo_model(person(clare)).
_xclingo_model(drive(gabriel)).
```

4.3.2 Reifing ASP rules

The reification mainly serves two purposes. On the one hand, it must retain the information about the rules themselves. This is, identifying each rule, recording its head and body, and representing the cause-effect relations that are assumed by `xclingo` from the rules of the original program. On the other hand, it must represent each `xclingo` annotation accordingly.

We will start discussing what the reification does not need to do. Since the approach is to start from an already computed model of the original program, and only to obtain explanations for that particular model, there is no need to take into account any part of the original program that may remove models or select among them. These are sentences that may remove invalid models such that constraints or that apply an ordering between them such that `#minimize`. During the preprocessing step, this kind of sentences are ignored. In the case of constraints, any model that `xclingo` explains is already a valid model of the original program, so considering the constraints will have no effect. Note that, this is the case of the default `xclingo` process and not the special case of explaining an unsatisfiable problem. That special case will be discussed later. In the case of `#minimize` sentences, `xclingo` will explain each model as it is obtained from `clingo` minimization until it reaches the minimum. Considering the original `minimize` sentences is not needed to explain an already obtained model. `#show` sentences are also ignored, that is, those that control which atoms are displayed in `clingo` output. Other elements do not need to be reified but they are included in the reification. This is the case of `#const` sentences, *theory atom* definitions or anything that does not fall in the category of *Rule* under the `clingo` internal representation of the program.

We will now discuss the aspect of the `xclingo` reification for rules and annotations. Note that the translation of an arbitrary program can be obtained for any program by using the `-output=translation` option like in Command 4.14.

```
xclingo example.lp -output=translation
```
(Command 4.14)

First, for any piece of ASP code that is not ignored by the preprocessor, `xclingo` will include the original version in a comment before the actual translation. After that, the translation of the element can be composed of 1 or more rules that fulfill different purposes. While discussing the translation, it is important to bear in mind that each rule in the program will have an orderly numbered identifier. This identifier is indeed, the label corresponding to the rule as explained in Chapter 3.

The first important predicate used to represent the reification is:

```
_xclingo_sup(RuleId, DisjunctionId, Head, (Var1, Var2, ..., VarN) )
```

It represents that the rule with id `RuleId`, and its atom `Head` are supported by the current model in the same sense as 3.3, from the encoding provided in ection 3.4 of Chapter 3. In the case the head of the rule is disjunctive, several instances would be generated each one incrementing the `DisjunctionId` argument to differentiate between them. The third argument `Head` represents the corresponding disjunctive atom from the head. Finally, the last argument represents the sequence of all the free variables used in the body of the rule.

For usual rules, at least a *support rule* is always generated where its head is an `_xclingo_sup/4` atom. In the case of disjunctive rules, a different support rule will be generated for each one of them by incrementing the third argument (`DisjunctionId`) as explained before. Following the meaning of the predicate `_xclingo_sup/4`, support rules are intended to be enabled when the original rule is supported by the considered model. To this aim, the body of support rules consists of `_xclingo_model/1` literals wrapping all the positive literals in the body of the original rule. For instance, the rules,

```
drive(gabriel).
punish(P) :- drive(P); alcohol(P,A); A > 30; person(P).
```

would be translated into the following support rules:

```
_xclingo_sup(1,0,drive(gabriel),()).
_xclingo_sup(2,0,punish(P),(P,A)) :- _xclingo_model(drive(P)),
          _xclingo_model(alcohol(P,A)), A > 30, _xclingo_model(person(P)).
```

In this way, for each possible set of values that the variables may take that meet the condition in the body, a support rule will derive a different `_xclingo_sup/4` atom. These different derivations will differ only in the values taken by the variables in the fourth argument.

The second predicate that is important for the reification is

```
_xclingo_depends(SupportAtom, (Lit1, Lit2, ..., LitN))
```

This predicate models the cause-effect dependencies between the head of a rule and the positive literals from the body. The first argument is the particular non-ground `_xclingo_sup/4` atom for the rule we are modeling the dependencies for. The second one is the sequence of the non-ground positive atoms in the body of the rule. To compute such cause-effect dependencies in terms of the current model, the preprocessor generates dependency rules for every original rule that is not a fact. The head is an atom of `_xclingo_depends` predicate, and the body is the corresponding `_xclingo_sup/4` literal. For instance, see now the complete translation for the previously translated program, which now includes a dependency rule.

```
_xclingo_sup(1,0,drive(gabriel),()).
_xclingo_sup(2,0,punish(P),(P,A)) :- _xclingo_model(drive(P)),
          _xclingo_model(alcohol(P,A)), A > 30, _xclingo_model(person(P)).
_xclingo_depends(_xclingo_sup(5,0,punish(P),(P,A)),(drive(P);alcohol(P,A);person(P))) :-
    _xclingo_sup(5,0,punish(P),(P,A)).
```

In plain words, dependency rules mean that we take for granted the cause-effect dependencies from the body of a rule if the rule is supported by the model.

In fact, if we add some atoms from a model,

```
_xclingo_model(drive(gabriel)). _xclingo_model(person(gabriel)). _xclingo_model(alcohol(gabriel,
    40)).
_xclingo_model(drive(robert)).  _xclingo_model(person(robert)).  _xclingo_model(alcohol(robert, 35)
    ).

_xclingo_sup(1,0,drive(gabriel),()).
_xclingo_sup(2,0,punish(P),(P,A)) :- _xclingo_model(drive(P)),
          _xclingo_model(alcohol(P,A)), A > 30, _xclingo_model(person(P)).
_xclingo_depends(_xclingo_sup(2,0,punish(P),(P,A)),(drive(P);alcohol(P,A);person(P))) :-
    _xclingo_sup(2,0,punish(P),(P,A)).
```

and we solve with clingo we obtain the following

```
_xclingo_model(drive(gabriel)) _xclingo_model(person(gabriel)) _xclingo_model(alcohol(gabriel,40))
_xclingo_model(drive(robert))  _xclingo_model(person(robert))  _xclingo_model(alcohol(robert,35))
_xclingo_sup(1,0,drive(gabriel),())
_xclingo_sup(2,0,punish(gabriel),(gabriel,40))
_xclingo_sup(2,0,punish(robert),(robert,35))
_xclingo_depends(_xclingo_sup(2,0,punish(gabriel),(gabriel,40)),person(gabriel))
_xclingo_depends(_xclingo_sup(2,0,punish(robert),(robert,35)),person(robert))
_xclingo_depends(_xclingo_sup(2,0,punish(gabriel),(gabriel,40)),alcohol(gabriel,40))
_xclingo_depends(_xclingo_sup(2,0,punish(robert),(robert,35)),alcohol(robert,35))
_xclingo_depends(_xclingo_sup(2,0,punish(gabriel),(gabriel,40)),drive(gabriel))
_xclingo_depends(_xclingo_sup(2,0,punish(robert),(robert,35)),drive(robert))
```

As we can see, we automatically get different dependencies for `punish(gabriel)` and for `punish(robert)`.

Note how in the head of generated Dependency rules, the second argument is a tuple where the positive literal from the body of the rule are separated by ';'. This clingo syntax construct is named *pool*, and it is a way to abbreviate several lines of code in one. In the following rule for instance,

```
_xclingo_depends(_xclingo_sup(2,0,punish(P),(P,A)),(drive(P);alcohol(P,A);person(P))) :-
    _xclingo_sup(2,0,punish(P),(P,A)).
```

Instead of repeating the rule three times varying the second argument with the different body literals (`drive(P)`, `alcohol(P, A)` and `person(P)`). We just instead use a pool to abbreviate the three rules in one. The latter code is equivalent to this:

```
_xclingo_depends(_xclingo_sup(2,0,punish(P),(P,A)),(drive(P)))     :- _xclingo_sup(2,0,punish(P),(P
    ,A)).
_xclingo_depends(_xclingo_sup(2,0,punish(P),(P,A)),(alcohol(P,A))) :- _xclingo_sup(2,0,punish(P),(P
    ,A)).
_xclingo_depends(_xclingo_sup(2,0,punish(P),(P,A)),(person(P)))    :- _xclingo_sup(2,0,punish(P),(P
    ,A)).
```

For each generated support rule (each one with its unique pair of `RuleId` and `DisjunctionId`), one dependency rule is generated. For instance, when translating a disjunctive rule like this one

```
resist(gabriel), obey(gabriel) :- stopped(gabriel).
```

note how each generated support rule (each one with its own disjunctive id 0 and 1) has its own dependency rule.

```
% resist(gabriel); obey(gabriel) :- stopped(gabriel).
_xclingo_sup(6,0,resist(gabriel),()) :- _xclingo_model(stopped(gabriel)).
_xclingo_depends(_xclingo_sup(6,0,resist(gabriel),()),stopped(gabriel)) :- _xclingo_sup(6,0,resist(
    gabriel),()).
_xclingo_sup(6,1,obey(gabriel),()) :- _xclingo_model(stopped(gabriel)).
_xclingo_depends(_xclingo_sup(6,1,obey(gabriel),()),stopped(gabriel)) :- _xclingo_sup(6,1,obey(
    gabriel),()).
```

This happens as well with choice rules, although in this case, the dependencies are a little bit more tricky to model. Consider the following choice rule,

```
1{
    pos(NewPos, T): NewPos=PrevPos+1, pos(PrevPos, T-1);
    pos(NewPos, T): NewPos=PrevPos-1, pos(PrevPos, T-1);
    pos(NewPos, T): NewPos=PrevPos+5, springboard(PrevPos), pos(PrevPos, T-1)
}1 :- timestep(T), T>0.
```

where an agent decides her new position based on the current one, being able to move one position to the right or the left or having the option to jump five positions to the right if there is a springboard in its current position. Depending on the choice that we consider, the dependencies for the derived atoms change. As it is part of the positive body of the rule, the literal `timestep(T)` will be treated as a valid dependency disregarding the choice. However, for each conditional atom, the positive part of the condition is also a dependency. This means that, in this case, all choices have a dependency with `pos(PrevPos, T-1)`, and that the springboard choice has an extra dependency with `springboard(PrevPos)`. Furthermore, the support rule must also change for the third option, to include the springboard condition, as it is a necessary condition for the third choice to derive anything. To sum up, the complete translation of the original choice rule above would be the following:

```
% 1 <= { pos(NewPos,T): NewPos = (PrevPos+1), pos(PrevPos,(T-1)); pos(NewPos,T): NewPos = (PrevPos
     -1), pos(PrevPos,(T-1)); pos(NewPos,T): NewPos = (PrevPos+5), springboard(PrevPos), pos(
     PrevPos,(T-1)) } <= 1 :- timestep(T); T > 0.
_xclingo_sup(1,0,pos(NewPos,T),(T,PrevPos,NewPos)) :- NewPos = (PrevPos+1); _xclingo_model(pos(
     PrevPos,(T-1))); _xclingo_model(timestep(T)); T > 0.
_xclingo_depends(_xclingo_sup(1,0,pos(NewPos,T),(T,PrevPos,NewPos)),(pos(PrevPos,(T-1));timestep(T)
     )) :- _xclingo_sup(1,0,pos(NewPos,T),(T,PrevPos,NewPos)).
_xclingo_sup(1,0,pos(NewPos,T),(T,PrevPos,NewPos)) :- NewPos = (PrevPos-1); _xclingo_model(pos(
     PrevPos,(T-1))); _xclingo_model(timestep(T)); T > 0.
_xclingo_depends(_xclingo_sup(1,0,pos(NewPos,T),(T,PrevPos,NewPos)),(pos(PrevPos,(T-1));timestep(T)
     )) :- _xclingo_sup(1,0,pos(NewPos,T),(T,PrevPos,NewPos)).
_xclingo_sup(1,0,pos(NewPos,T),(T,PrevPos,NewPos)) :- NewPos = (PrevPos+5); _xclingo_model(
     springboard(PrevPos)); _xclingo_model(pos(PrevPos,(T-1))); _xclingo_model(timestep(T)); T > 0.
_xclingo_depends(_xclingo_sup(1,0,pos(NewPos,T),(T,PrevPos,NewPos)),(springboard(PrevPos);pos(
     PrevPos,(T-1));timestep(T))) :- _xclingo_sup(1,0,pos(NewPos,T),(T,PrevPos,NewPos)).
```

On it, 6 rules were generated: one support and one dependency rule for each possible choice. Each one of them captures the correct support and causal dependencies between the atoms.

Something similar happens when translating aggregates. Consider for example the following rule,

```
traffic_incidents(N) :- N = #count { P: alcohol(P,A), drive(P), A > 30; P: park(P,Z),
     no_parking_zone(Z) }, person(P).
```

where we count the number N of persons P that either drove drunk or parked in a no-parking zone. Considering when this rule is supported, the only necessary literal is `person(P)`. In the case no person has committed any of both offences, the counter N will take the value 0 and we derive the atom. This means that this rule only needs to generate one support rule, checking if `person(P)` is in the model. Regarding the causal dependencies, the idea is that an atom is a cause when it contributes to the total count. For instance, if we find out that `gabriel` drove drunk, we have to increase the counter, and *both* atoms `alcohol(gabriel, A)` (having A some ground value) and `drive(gabriel)` become causes of the derived `traffic_incidents` atom. The same happens for `park` and `no_parking_zone` atoms, for instance, when `clare` parks in a no-parking zone. If a third person `daniel` also parked at an illegal zone, another two atoms become causes. In short, for any way to verify one of the conditional atoms within the aggregate, all the atoms used in the condition become causes for the head of the rule. This means that aggregates will produce a dependency rule for each conditional atom. The following code is the translation for the previously shown rule.

```
% traffic_incidents(N) :- N = #count { P: alcohol(P,A), drive(P), A > 30; park(P,Z),no_parking_zone
     (Z) }; person(P).
_xclingo_sup(1,0,traffic_incidents(N),(P,N)) :- N = #count { P: _xclingo_model(alcohol(P,A)),
     _xclingo_model(drive(P)), A > 30; park(P,Z),no_parking_zone(Z) }; _xclingo_model(person(P)).
_xclingo_depends(_xclingo_sup(1,0,traffic_incidents(N),(P,N)),person(P)) :- _xclingo_sup(1,0,
     traffic_incidents(N),(P,N)).
_xclingo_depends(_xclingo_sup(1,0,traffic_incidents(N),(P,N)),(alcohol(P,A);drive(P))) :-
     _xclingo_model(alcohol(P,A)); _xclingo_model(drive(P)); A > 30; _xclingo_sup(1,0,
     traffic_incidents(N),(P,N)).
```

Remark on explaining aggregates

Although it is clear to us that every possible combination that triggers a conditional atom should be considered a cause, that does not mean that a proper explanation of any aggregate should include all of them. In the case of `#count` aggregates, where every combination contributes to the sum, it seems safe to include them all as a cause. But in the case of monotone aggregates, for instance, this is not so clear. Following what we explained in Example 4.8, providing good explanations for the aggregates would probably require some sort of minimization of the set of causes that still comply with the conditions over the aggregate. Therefore, generating a choice as well as a `#minimize` sentence in the translation seems to be the way to go.

However, it is future work to see if this strategy would serve to explain other aggregates that follow different semantics.

4.3.3 Show Trace annotations and *to-explain atoms*

Computing the support graphs is possible now that the already shown ASP translation can figure out which rules are supported and the causal dependencies among the atoms. In Chapter 3, we explained how to obtain the support graphs given the program and a model. However, it is important to note that `xclingo`, in contrast, is oriented towards explaining certain atoms. These atoms are specified by the user using `show_trace` annotations. From now on, we will refer to the atoms the user has queried as *to-explain atoms*.

To represent such atoms we use the predicate `_xclingo_show_trace/1`, where its only argument is a to-explain atom. Any atom belonging to this predicate is of course obtained from the translation of `show_trace` annotations. Whenever an annotation of this kind is found, it is translated into a simple rule, which checks the atoms from the model to derive some to-explain atoms. For instance, the following annotation,

```
%!show_trace {sentence(P,S)} :- age(P,A), A<18.
```

where the user asks for sentences for persons P under 18 years old, it is translated into the following rule

```
_xclingo_show_trace(sentence(P,S)) :- _xclingo_model(sentence(P,S)); _xclingo_model(age(P,A)); A <
    18.
```

As it can be seen, the literals in the body of the annotation are replaced by new literals using `_xclingo_model/1` predicate to wrap up the atoms from the original literals. The conditions over the variables remain untouched. In this way, `_xclingo_show_trace/1` tells us which atoms should be explained for the given model.

Knowing which atoms the user is interested in before actually computing the graphs offers some computational advantages. For instance, since we know the dependencies between the atoms (via `_xclingo_depends/2`), we can start from the atoms that we want to explain and identify which atoms are relevant for explaining such atoms by traversing the dependencies. These relevant atoms are represented in the encoding by the `_xclingo_relevant/1` predicate. The rules from where we obtain the relevant atoms are the following:

```
% Marks relevant atoms of the program, concerning the atoms that must be explained.
_xclingo_relevant(ToExplainAtom) :- _xclingo_show_trace(ToExplainAtom).
_xclingo_relevant(Cause) :- _xclingo_relevant(Effect), _xclingo_depends(_xclingo_sup(R, D, Effect,
    Vars), Cause).
```

They can be interpreted in plain language as (1) any to-explain atom is relevant; and (2) any atom that is identified as a `Cause` of another relevant atom `Effect` by some rule `R`, is also relevant.

4.3.4 Building Support Graphs

In Definition 6 we have explained how, in a particular support graph, only one rule can be responsible for explaining a particular atom. This condition is indeed reflected by the fact that *the λ function must be injective*. This means that, in a single support graph, one supported rule can be used to justify an atom. The predicates `_xclingo_fbody/4` and `_xclingo_f/4` are used to select which rules are used in each support graph.

The four arguments of both predicates are analogous to the arguments from the already explained predicate `_xclingo_sup/4`. In the case of `_xclingo_fbody/4`, the `fbody` terms come from a "fireable" body. In plain words, an arbitrary atom

```
_xclingo_fbody(RuleId, DisjunctionId, Head, (Var1, Var2, ..., VarN) )
```

means that the rule `RuleId`, is eligible to be selected to explain the atom `Head` in the current support graph, with some particular ground values of the free variables in the body of the rule. On the other hand, the `f` of `_xclingo_f/4` comes from *fired*, and an atom

```
_xclingo_f(RuleId, DisjunctionId, Head, (Var1, Var2, ..., VarN) )
```

means that the rule `RuleId`, has been selected to explain the atom `Head` in the current support graph, again, with some particular values for the variables. The `DisjunctionId` variable identifies which atom the predicate talks about among the disjunctive atoms in the head of rule `RuleId`.

Knowing the meaning of these two predicates, the choice rule responsible for selecting which rules explain which atoms easily follows.

```
1{_xclingo_f(RuleID, D, Atom, Vars) : _xclingo_fbody(RuleID, D, Atom, Vars)}1 :- _xclingo_relevant(
     Atom).
```

Intuitively, the rule can be read as: *for any relevant atom, select one and only one rule to fire it (or to explain it) from the eligible rules*. Whenever we have several rules that may fire or explain the atom, we generate a different graph. This directly relates with rules 3.10 and 3.12 of the encoding provided in Section 3.4 of Chapter 3 Besides the cardinality constraint imposed, mimics the effect of rules 3.8 and 3.8 also from the original encoding.

Before explaining how we decide when a rule is fireable or eligible, let us introduce a third predicate `_xclingo_f_atom/1`. This predicate simply acts as a shortcut to know if an atom has a rule that explains it. The corresponding rule following this definition is

```
_xclingo_f_atom(Atom) :- _xclingo_f(_, _, Atom, _).
```

Now, the definition of a fireable rule follows two conditions. First, that any supported fact rule that features a relevant atom is fireable. This definition is specified in the following ASP rule:

```
_xclingo_fbody(RuleID, D, Atom, Vars) :- _xclingo_relevant(Atom), _xclingo_sup(RuleID, D, Atom,
     Vars), not _xclingo_depends(_xclingo_sup(RuleID, D, _, _), _).
```

Note how the way to specify when a rule is a fact is by checking whether it does not have any causal dependency. The second condition says that a non-fact rule is fireable if all its dependencies can fired or explained by some other rule. The rule implementing this is the following:

```
_xclingo_fbody(R, D, Atom, Vars) :-
    _xclingo_sup(R, D, Atom, Vars),
    _xclingo_f_atom(Cause) : _xclingo_depends(_xclingo_sup(R, D, Atom, Vars), Cause).
```

The two last discussed rules above relate with rule 3.5 of the encoding provided in Section 3.4 of Chapter 3

In this way, `_xclingo_fbody/4` and `_xclingo_f/4` form a recursive definition that decides the labeling of the atoms in the graph (i.e. which rules are selected to explain the atoms). This definition focuses on relevant atoms, starts firing rules from the facts (i.e. those having an empty body) and recursively selects new fireable rules until every relevant atom is explained by some rule.

However, this is not enough to comply with the *injectiveness* condition of the λ function from Definition 6. To do so, we still have to deal with disjunction. The choice rule which decides which rules are fired forbids that an atom be explained by two rules, however, it does not forbid two different atoms being explained by the same rule. To prevent this, we introduce the following constraint:

```
:- _xclingo_f(R,D1,_,_), _xclingo_f(R,D2,_,_), D1!=D2.
```

Which ensures only one of the disjunctive heads of a rule is fired by it.

At this point, we know the nodes from the graph which are given by `_xclingo_f_atom/1` or `_xclingo_relevant/1`. The λ function from Definition 6 (i.e. which rule explains each atom in the graph) is given by `_xclingo_f/4`. The only remaining thing we miss in building the graph are the edges, which can be defined in the following way:

```
_xclingo_direct_cause(RuleID, Effect, Cause) :- _xclingo_f(RuleID, DisID, Effect, Vars),
     _xclingo_depends(_xclingo_sup(RuleID, DisID, Effect, Vars), Cause).
```

Note that having the first argument `RuleId` is not necessary to build the graph, as having the edge vertices `Effect` and `Cause` completely defines it. However, this is useful when filtering the graph via `mute` and `mute_body` annotations, as we will explain later in this chapter.

The code explained up to this point corresponds with one of the core ASP programs of `xclingo` called `xclingo_fired.lp`, which is completely listed in Appendix B.1.

Now let us consider an example to demonstrate how we can obtain the support graphs for any ASP program. Recall again Program 4.1, but including also the following `show_trace` annotation:

```
%!show_trace{sentence(gabriel, S)}.
```

This means that we will only query explanation for `gabriel`'s sentences. First, we need a model to explain. As we have already seen, the program has a total of 3 answer sets. We will stick with one of them, and encode the model as we have explained for the translation. The model we will explain, which is already properly encoded, is listed below:

```
% model3.lp

_xclingo_model(alcohol(clare,5)). _xclingo_model(alcohol(gabriel,40)).
_xclingo_model(person(gabriel)).  _xclingo_model(person(clare)).
_xclingo_model(drive(gabriel)).   _xclingo_model(drive(clare)).
_xclingo_model(sentence(clare,innocent)).
_xclingo_model(punish(gabriel)).
_xclingo_model(resist(gabriel)).
_xclingo_model(sentence(gabriel,prison)).
```

Listing 4.1: Encoding for the third model of Program P 4.1, shown in Output 4.1

Note how it corresponds to the third model shown in Output 4.1, where `gabriel` can be punished both for driving drunk and for resisting authority. The translation for Program 4.1 extended with the previous `show_trace` annotation is listed below.

```
% dont_drive_drunk_translation.lp

% person(gabriel;clare).
_xclingo_sup(1,0,person(gabriel;clare),()).
% drive(gabriel).
_xclingo_sup(2,0,drive(gabriel),()).
% drive(clare).
_xclingo_sup(3,0,drive(clare),()).
% alcohol(clare,5).
_xclingo_sup(4,0,alcohol(clare,5),()).
% 1 <= { alcohol(gabriel,40); resist(gabriel) }.
_xclingo_sup(5,0,alcohol(gabriel,40),()).
_xclingo_sup(5,0,resist(gabriel),()).
% punish(P) :- drive(P); alcohol(P,A); A > 30; person(P).
_xclingo_sup(6,0,punish(P),(A,P)) :- _xclingo_model(drive(P)); _xclingo_model(alcohol(P,A)); A >
    30; _xclingo_model(person(P)).
_xclingo_depends(_xclingo_sup(6,0,punish(P),(A,P)),(drive(P);alcohol(P,A);person(P))) :-
    _xclingo_sup(6,0,punish(P),(A,P)).
```

```
% punish(P) :- resist(P); person(P).
_xclingo_sup(7,0,punish(P),(P,)) :- _xclingo_model(resist(P)); _xclingo_model(person(P)).
_xclingo_depends(_xclingo_sup(7,0,punish(P),(P,)),(resist(P);person(P))) :- _xclingo_sup(7,0,punish
     (P),(P,)).
% sentence(P,innocent) :- person(P); not punish(P).
_xclingo_sup(8,0,sentence(P,innocent),(P,)) :- _xclingo_model(person(P)); not _xclingo_model(punish
     (P)).
_xclingo_depends(_xclingo_sup(8,0,sentence(P,innocent),(P,)),person(P)) :- _xclingo_sup(8,0,
     sentence(P,innocent),(P,)).
% sentence(P,prison) :- punish(P).
_xclingo_sup(9,0,sentence(P,prison),(P,)) :- _xclingo_model(punish(P)).
_xclingo_depends(_xclingo_sup(9,0,sentence(P,prison),(P,)),punish(P)) :- _xclingo_sup(9,0,sentence(
     P,prison),(P,)).
% &show_trace { sentence(gabriel,S) }.
_xclingo_show_trace(sentence(gabriel,S)) :- _xclingo_model(sentence(gabriel,S)).
```

Listing 4.2: Translation of Program P 4.1 extended with an extra `show_trace` annotation

These two pieces of ASP code, put together with the program explained up to this point (completely depicted in Listing B.1 in Appendix B) will compute the corresponding explanations if solved. This will, however, only compute the subgraph that is relevant for explaining the sentence for `gabriel` as explained before. Thus, by running Command 4.15,

```
clingo 0 model3.lp dont_drive_drunk_translation.lp xclingo_fired.lp
```
(Command 4.15)

we obtain the models representing the explanations listed below.

```
Answer: 1
_xclingo_direct_cause(7,punish(gabriel),resist(gabriel))
_xclingo_direct_cause(7,punish(gabriel),person(gabriel))
_xclingo_direct_cause(9,sentence(gabriel,prison),punish(gabriel))
Answer: 2
_xclingo_direct_cause(6,punish(gabriel),drive(gabriel))
_xclingo_direct_cause(6,punish(gabriel),alcohol(gabriel,40))
_xclingo_direct_cause(6,punish(gabriel),person(gabriel))
_xclingo_direct_cause(9,sentence(gabriel,prison),punish(gabriel))
satisfiable
```

We only show the `_xclingo_direct_cause/3` atoms for brevity. As we can see, it finds two explanations. One can manually draw both graphs and check that they match with the explanations obtained by `xclingo` for the third model when running Command 4.16

```
xclingo -n 0 0 dont_drive_drunk.lp -auto-tracing=all
```
(Command 4.16)

```
Answer: 3
##Explanation: 3.1
    *
    |__sentence(gabriel,prison)
    |  |__punish(gabriel)
    |  |  |__drive(gabriel)
    |  |  |__alcohol(gabriel,40)
    |  |  |__person(gabriel)

##Explanation: 3.2
    *
    |__sentence(gabriel,prison)
    |  |__punish(gabriel)
    |  |  |__person(gabriel)
    |  |  |__resist(gabriel)
```

4.3.5 Filtering support graphs

We will now explain how to add the annotations to the translation in order to filter the explanations (i.e. performing the *edge pruning* as well the *node forgetting* operations over the graph) and display customized text traces. First, we start discussing how we translate `mute` and `mute_body` annotations to process the pruning operations over the final graph. As Definition 9 tells, the prune operations operate over a set of atoms for which we want to delete the incoming edges and a set of labels such that we will delete the outgoing edges of the atoms explained by those rules. The predicates the `mute` and `mute_body` translate into, respectively serve those purposes. For muting atoms with `mute` annotations we use the `_xclingo_muted/1` predicate, where the only argument of the predicate is a muted atom. The translation works exactly as the translation for the `show_trace` annotation, explained before. For instance, the following `mute` annotation,

```
%!mute {person(P)}.
```

is translated into the following rule

```
_xclingo_muted(person(P)) :- _xclingo_model(person(P)).
```

In the case of `mute_body` annotations, the `_xclingo_muted_body/1` predicate is used, where the only argument of the predicate is the identifier (i.e. label) of the muted rule. To illustrate the translation of `mute_body` annotations, consider the following annotation

```
%!mute_rule.
person(P) :- student(P).
```

for whose translation would be the following rules

```
_xclingo_muted_body(2).
_xclingo_sup(2,0,person(S),(S,)) :- _xclingo_model(student(S)).
_xclingo_depends(_xclingo_sup(2,0,person(S),(S,)),student(S)) :- _xclingo_sup(2,0,person(S),(S,)).
```

where `2` is the label of the rule affected by the `mute_body`.

Adding this into the translation will provide us with both sets of muted atoms and muted rules, that we need to perform the pruning operation over the graph. But to represent the new resulting graph after the pruning operation (and also after the forgetting operation), we will use three new predicates. The first predicate is `_xclingo_graph/1` which identifies the graphs. Namely, we have `_xclingo_graph(pruned)` which identifies the graph after the pruning operation and `_xclingo_graph(forgotten)` which identifies the graph after the forgetting operation. The second predicate we introduce is `_xclingo_node(Atom,` Graph) where Atom is the atom within the node and Graph references either the prunned graph or the forgotten. Finally, we introduce _xclingo_edge((Effect, Cause), Graph), where Effect and Cause depict the directed edges of graph Graph, which again can be one of the two graphs.

Knowing this, recall predicates _xclingo_show_trace/1 and _xclingo_direct_cause/3 explained in the previous section and consider the ASP code below used to compute the pruning graph from the original support graph.

```
_xclingo_node(ToExplainAtom, pruned) :- _xclingo_show_trace(ToExplainAtom), not _xclingo_muted(
     ToExplainAtom).
_xclingo_edge((Caused, Cause), pruned) :-
    _xclingo_node(Caused, pruned),
    _xclingo_direct_cause(RuleID, Caused, Cause),
    not _xclingo_muted(Cause),
    not _xclingo_muted_body(RuleID).
_xclingo_node(Atom, pruned) :- _xclingo_edge((_, Atom), pruned).
```

In plain words, we can interpret the rules as (1) any non-muted relevant atom is a node of the pruned graph; (2) any edge from the original graph that aims to a node shared with the pruned graph is included, if the atom at the source of the edge is neither a muted atom nor it was fired by a muted rule; and (3), any atom at the source of an included edge is as well a node of the pruned graph. Again, this consists of a recursive definition that starts from the to-explain atoms and includes new atoms from the original graph whenever they or their rules are not muted

Now, we will discuss the translation of trace and trace_rule operations, which should give us the tools to compute the forgotten (and final) graph, but also help us to keep track of text traces. In this case, both annotations share the same predicate for representing the labels, which is _xclingo_label/2. The first argument is the atom that is being traced, and the second is the user-defined text trace. In the case of trace annotations, the translation is very similar to the translation of show_trace and mute annotations. Consider the translation of the following annotation,

```
%!trace {alcohol(P,A), "% alcohol's level is %",P,A} :- alcohol(P,A), A>30.
```

shown below:

```
_xclingo_label( alcohol(P,A), @label("% alcohol's level is %", (P, A,)) ) :- _xclingo_model(alcohol
     (P,A)), A>30.
```

Again, the literals in the body of the annotation become wrapped by the _xclingo_model/1 atoms. The only new concept included in this translation is the use of the @label predicate. This is a special type of predicate that can be controlled via an external, Python-coded, object named *Context*. During solving, whenever the rule is derived or grounded, the corresponding Python code will be queried and, after some processing, the atom will be replaced by some ASP code. In this case, all @label predicates are replaced by the text trace, after linking the ground values of the variables to the text.

The translation changes a bit for trace_rule annotations. In this case, the atoms derived from the head of a rule must be traced only if that rule was fired or selected to explain the atom in the current support graph. This means that we need to somehow link the derivation of some _xclingo_label/2 atom to the selection of the corresponding rule. This is done by using an _xclingo_f/4 literal in the body of the translated rule. For instance, consider the following traced rule,

```
%!trace_rule {"% drove drunk (over 30mg/l)", P}.
punish(P) :- drive(P), alcohol(P,A), A>30, person(P).
```

and its translation is shown below:

```
% punish(P) :- drive(P); alcohol(P,A); A > 30; person(P).
_xclingo_sup(1,0,punish(P),(P,A)) :- _xclingo_model(drive(P)); _xclingo_model(alcohol(P,A)); A >
     30; _xclingo_model(person(P)).
_xclingo_depends(_xclingo_sup(1,0,punish(P),(P,A)),(drive(P);alcohol(P,A);person(P))) :-
     _xclingo_sup(1,0,punish(P),(P,A)).
_xclingo_label(Head,@label("% drove drunk (over 30mg/l)",(P,))) :- _xclingo_f(1,DisID,Head,(P,A)).
```

Note how the identifier of the rule used in the literal in the body matches the identifier used in the rest of the translation for that rule. The text trace is processed in the same way as we explained for `trace` annotations.

Now, once we have an support graph and we have pruned it following `mute` and `mute_body` annotations, `_xclingo_label/2` atoms will tell us which atoms are traced, providing us with the ability to perform the node forgetting operation over our graph and obtaining the filtered final explanation. To this aim, we introduce the predicate `_xclingo_visible/1` whose atoms come from `_xclingo_label/2` as in the following rule:

```
_xclingo_visible(X) :- _xclingo_node(X, pruned), _xclingo_label(X, _).
```

In other words, any traced atom that is a node of the `pruned` graph is visible. From that, we now need to create new edges that omit the forgotten atoms and connect the visible ones. For that purpose, we introduce predicate `_xclingo_skip` and `_xclingo_reach/2`, where `_xclingo_skip/2` identify edges that should be removed, and `_xclingo_reach(X,Y)` means that we can traverse the graph from node `X` to `Y` using only forgotten nodes in the `pruned` graph. The definition follows the rules below:

```
_xclingo_skip(X, Y) :- _xclingo_edge((X, Y), complete_explanation), not _xclingo_visible(X).
_xclingo_skip(X, Y) :- _xclingo_edge((X, Y), complete_explanation), not _xclingo_visible(Y).

_xclingo_reach(X, Z) :- _xclingo_skip(X, Z).
_xclingo_reach(X, Z) :- _xclingo_reach(X, Y), _xclingo_skip(Y, Z), not _xclingo_visible(Y).
```

Finally, we compute the `forgotten` graph using the definitions from the rules below:

```
_xclingo_node(Caused, forgotten) :- _xclingo_visible(Caused).
_xclingo_node(ToExplainAtom, forgotten) :- _xclingo_show_trace(ToExplainAtom).
_xclingo_edge((Caused, Cause), forgotten) :- _xclingo_edge((Caused, Cause), pruned), not
     _xclingo_skip(Caused, Cause).
_xclingo_edge((Caused, Cause), forgotten) :- _xclingo_reach(Caused, Cause), _xclingo_visible(Caused
     ), _xclingo_visible(Cause).
_xclingo_edge((ToExplainAtom, Cause), forgotten) :- _xclingo_reach(ToExplainAtom, Cause),
     _xclingo_visible(Cause), _xclingo_show_trace(ToExplainAtom).
```

From them, we can see that any to-explain atom or visible (i.e. traced) atom is a node of the graph. For the rules concerning the edges, we know that (1) we have to include any edge from the `pruned` graph that is not skippable (i.e. it does not exist a `_xclingo_skip/2` atom concerning that edge); (2) we have to form new edges between visible nodes which are reachable through skippable edges; and (3), we have to form edges that connect the original to-explain atoms to reachable nodes.

The whole ASP rules shown in this section are put together in the ASP file *xclingo_graph.lp* which is completely listed in Listing B.2 in Appendix B. When *xclingo_fired.lp* and *xclingo_graph.lp* are added to the translation of an annotated program, together with an encoded model, one can solve it and collect either the complete graph, the `pruned` graph or the `forgotten` graph. Besides, with each tuple of graphs, we obtain a set of `_xclingo_label/2` atoms which gives us the text traces for the traced atoms.

4.3.6 Relaxing Constraints for Explaining Unsatisfiability

For the explanation of unsatisfiable programs, little changes are made to the processing of the input program nor the explainer program. As we explained in Section 4.3.2, constraints are normally not translated. For obtaining an explanation when there is no answer set, we have explained how `xclingo` first relaxes any traced constraint and then applies a minimization to minimize the number of times a constraint is violated. In this section, we will discuss how these two steps are implemented to obtain explanations.

The first step is trying to repair the original annotated program to obtain some models. We only focus on cases where [1] the traced constraints are causing the unsatisfiability and so they are relaxed. Thus, an auxiliary head is generated for any traced constraint. This head consists of the predicate `_xclingo_violated_constraint(ConstraintId, (VAR1, VAR2, ...  , VARN))` where `ConstraintId` is an identifier for the particular constraint, and `VAR1`, `VAR2` and `VAR3` are the free variables used in the body of the constraint. The identifier is an incrementally generated number handled during this special translation and is independent of the rule identifiers (i.e. labels). All the free variables are listed in the head for the derived `_xclingo_violated_constraint` to distinguish between the different activations of these rules. Thanks to the auxiliary head, a particular atom records every time the body of a traced constraint is true. Take for instance the following traced constraint from Program 4.10.

```
%!trace_rule {"I can't reach vertex % from vertex 1",X}.
:- vtx(X), not reached(X).
```

After the relaxation step, it is translated into the following traced rule:

```
%!trace_rule {"I can't reach vertex % from vertex 1",X}.
_xclingo_violated_constraint(14,(X,)) :- vtx(X); not reached(X).
```

The minimization of the violated constraints now trivially follows. The following `#minimize` sentence is added to the original program:

```
#minimize{1,ID, Vars: _xclingo_violated_constraint(ID, Vars)}.
```

In this way, when solving the new program, each generated model (if any) will have progressively fewer `_xclingo_violated_constraint` atoms until reaching a minimum.

Keep in mind that these two steps are done before the solving and the translation of the program. This is, they aim to obtain models, not explanations. Assuming that we get a satisfactory program,

[1]For simplicity, we do not consider unsatisfiability due to odd loops through default negation, which would require a much more elaborated approach, but also, would assume a more technical knowledge from the user to which the explanation is targeted.

we only need to make a little change to the translation. This change is related, more precisely, to the processing of the `show_trace` annotations. As it was explained in Section 4.2.7, any custom `show_trace` annotation is ignored. In short, the `show_trace` annotations are not translated anymore, so they remain as mere comments in the code. Besides, the explanations of the constraints are automatically shown. For this purpose, a particular `show_trace` annotation that is shown below is added into the translation by `xclingo`.

```
_xclingo_show_trace(_xclingo_violated_constraint(ID, Vars)) :- _xclingo_model(
    _xclingo_violated_constraint(ID, Vars)).
```

The annotation queries any violation of any constraint, making `xclingo` to explain the `_xclingo_violated_constraint` atoms as usual.

4.4 Architecture and Design

Figure 4.3 shows the architecture of `xclingo`. The tool is modularly divided into big four independent components, with well-defined responsibilities that work together to generate explanations. *Preprocessing* and *Explaining* modules are Python code, while *Extensions* and *Xclingo LP* are pure ASP code. The *Preprocessing* module is a very large part of the tool which reifies the annotated logic program into the translation that was fully explained in sections from 4.3.1 to 4.3.3. The *Xclingo LP* module consists of 3 ASP files, responsible for computing the explanations from the translation and models of the original program. The specification encoded in such files was already explained in detail in sections from 4.3.4 and 4.3.5. The *Extensions* module is a set of ASP programs that are used to *extend* either the solving program or the explainer program to modify/increment the default functionality. Finally, the *Explaining* controls the explaining phase, collecting the models as well as the translation and also calls `clingo` for collecting back the explanations to finally bring them to the user.

The rest of the chapter is organized as follows. Section 4.4.1 explains the design behind the *preprocessing* module and explains how the translation is generated. Section 4.4.2 discusses the flow of the data from when the models of the original program are obtained until the explanations are displayed. Finally, Section 4.4.3 shows some *extensions* `xclingo` already uses by default and discusses the implementation of custom user-defined extensions. Note that *Xclingo LP* module was already described in detail in Section 4.3

4.4.1 Preprocessing module

The *Preprocessing* module is responsible for processing the original annotated program to produce the corresponding translation. The task consists of receiving the original annotated program as a String, parsing it to an *Abstract Syntax Tree* (AST) and finally returning a freshly new AST, which depicts the translation. To implement such functionality in a maintenance-friendly way, it has to face two main problems. (1) The problem of organizing the different translation strategies that the tool already considers (namely, the default one and the one for explaining unsatisfiable programs), and any new one that could arise. And (2), writing the translation code in a way that code-reusability is maximized. Figure 4.4 depicts the design of the module. The classes *PreprocessorPipeline* and *Preprocessor* together implement a *Pipeline* pattern, where a *Pipeline* object can register several preprocessors that will sequentially process the program. Each preprocessor assesses different purposes in the translation like

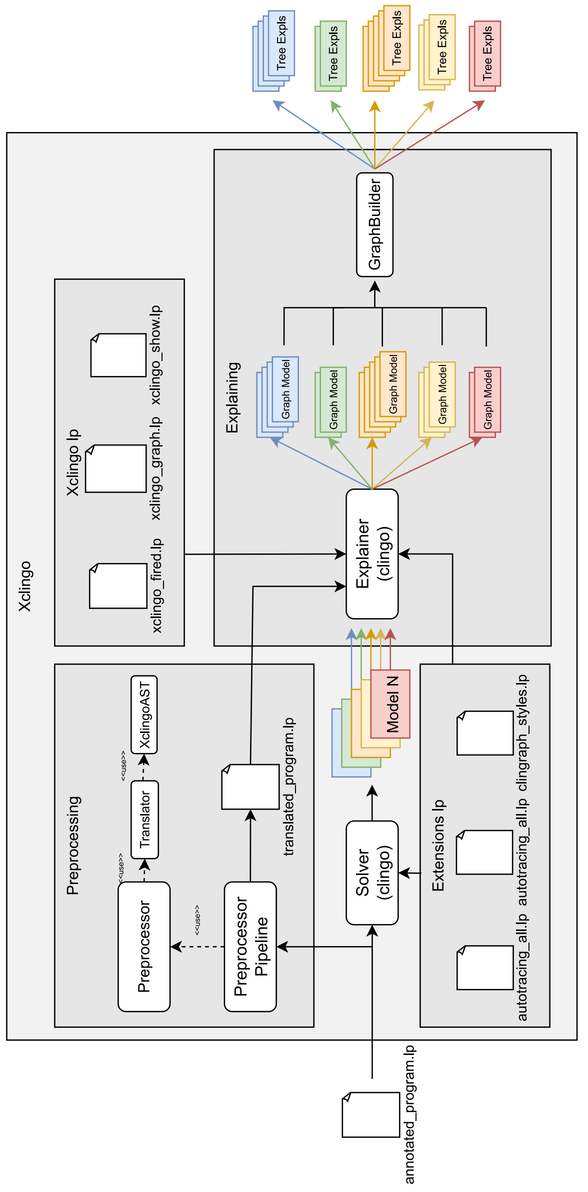


Figure 4.3: Architecture of `xclingo`.

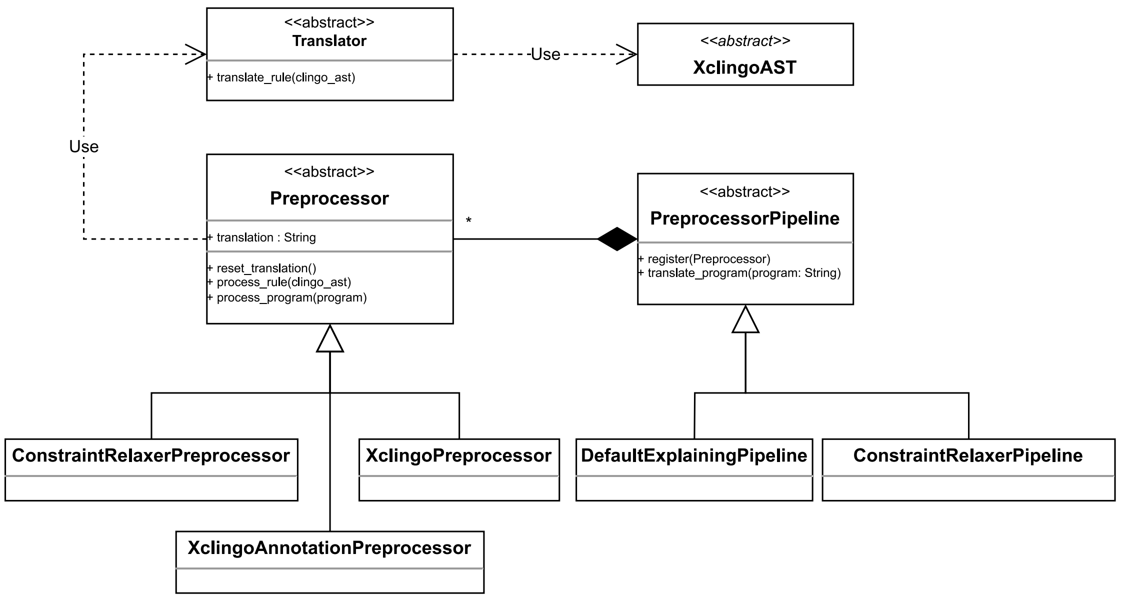

<<abstract>>
Translator
+ translate_rule(clingo_ast)
Use
<<abstract>>
XclingoAST
Use
<<abstract>>
Preprocessor
+ translation : String
+ reset_translation()
+ process_rule(clingo_ast)
+ process_program(program)
*
<<abstract>>
PreprocessorPipeline
+ register(Preprocessor)
+ translate_program(program: String)
ConstraintRelaxerPreprocessor
XclingoPreprocessor
DefaultExplainingPipeline
ConstraintRelaxerPipeline
XclingoAnnotationPreprocessor

Figure 4.4: Pipeline pattern for the *Preprocessing* module.

converting the annotations into a middle-step translation, relaxing the annotated constraints or directly translating the the rules. Thus, each pipeline registers different preprocessors in a different order, which leads to different results. A user could potentially implement and register custom preprocessors and/or pipelines to modify or extend the translation.

The default *Preprocessors* use `clingo`'s Python API to parse the program in a String form into a sequence of AST objects. These AST objects are the representation of the major structures from the annotated ASP program, such as rules, constraints, `#show` statements, etc. One by one, they process each AST, identify the type of expression they belong to, pass it to the corresponding *Translator* to generate the translation, and store this result, maintaining a record of the ongoing translation. The *XclingoAST* class implements a *Decorator* pattern, which is used to build the fresh translated ASTs (such as support Rules, dependency Rules and the rest of parts of the translation that are discussed in sections from 4.3.2 to 4.3.6) starting from the original ASTs. The *Translator* class acts as the client for this decorator, decoupling that knowledge from the *Preprocessor* who only sends and receives AST objects.

Some of the different types of rules built for the translation share common parts. For instance, the predicate used in the head of the support rules is the same one used for the body of dependency rules and the positive part of the body of the original rules is wrapped by the same predicate for `show trace annotations` and `trace annotations`. To keep the design clean, maximize code reuse and enhance maintainability, the *xclingoAST* decorator implementation makes use of multiple inheritance. A base component defines the basic functionality, which consists of translating an AST by separately building the head and body of the new rule. Figure 4.5 shows the design of the basic decorators. The base components are *XclingoAST* and *XclingoRule*, *Body* and *ReferenceLit* which define the interface that any other decorator must implement. For instance, the *ModelBody* decorator produces a body where the positive literals from the original rule are wrapped with a `_xclingo_model` predicate to ref-

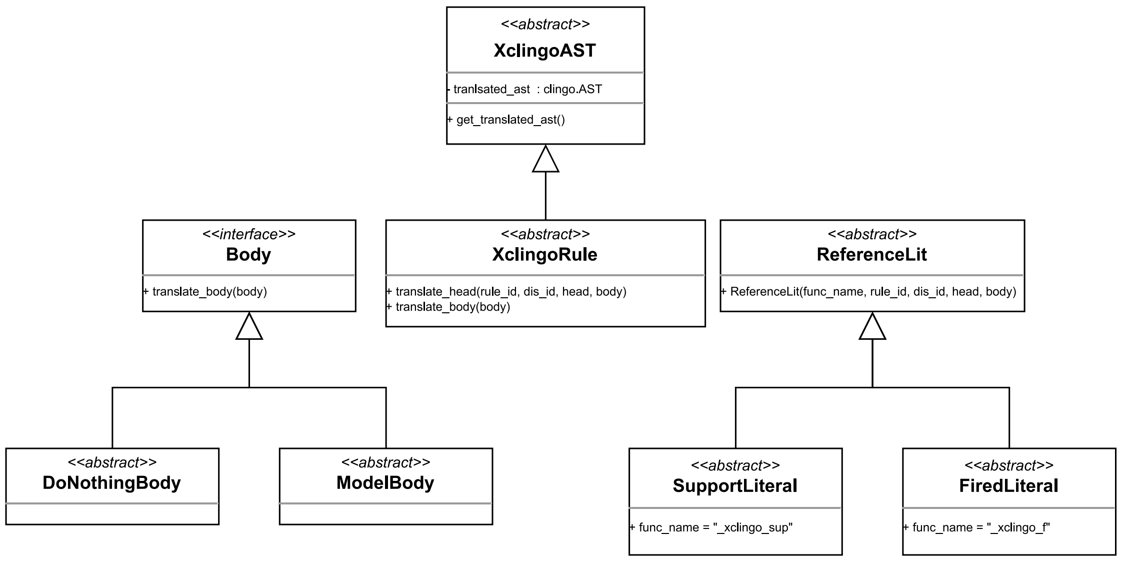


Figure 4.5: Base components of the *Decorator* pattern for designing the *XclingoAST* class.

erence the model that is being explained. Another example is the *SupportLiteral* decorator provides a way to build the `_xclingo_sup` predicate. These decorators are later reused to define complex rules. For instance, in Figure 4.6 the different types of rules that form the translation are defined by inheriting from the basic decorators. Each of the three components drawn here (*RelaxedConstraint*, *SupportsRule*

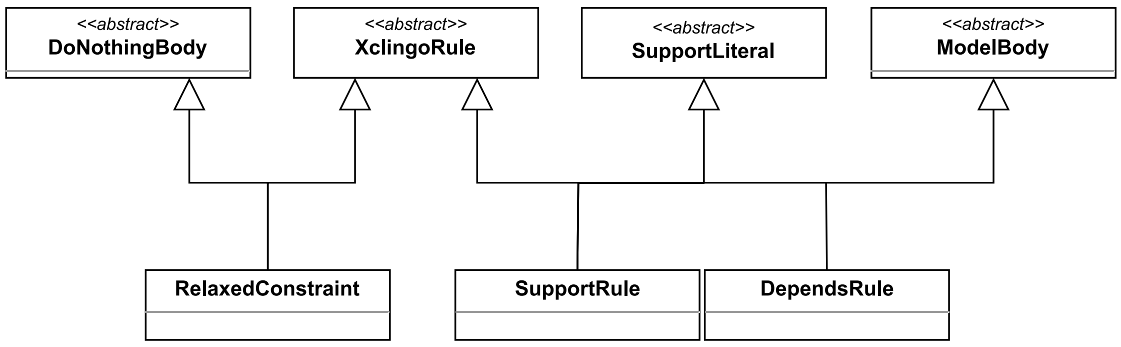


Figure 4.6: Decorated components for the main rules of the translation.

and *DependsRule*) inherit some common behavior from the basic decorators but define its additions as well. For instance, even though the latter two inherit from the same classes the translation they produce is a different kind of translated rule. However, in this way, we avoid repeating code and simplify both the design and the effort of adding new translations.

Figures 4.7 and 4.8 show the decorators and the components used to build the different translations for the annotations.

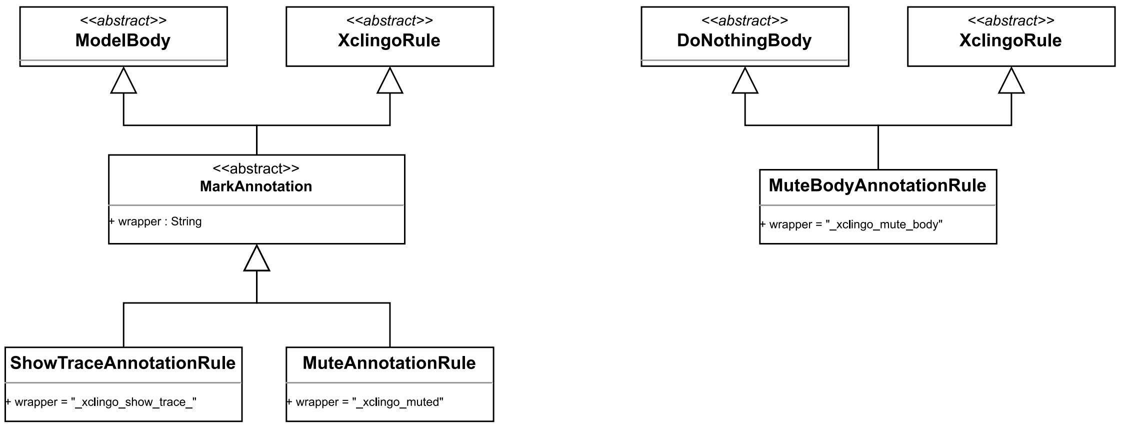


Figure 4.7: Decorated components for some annotation rules.

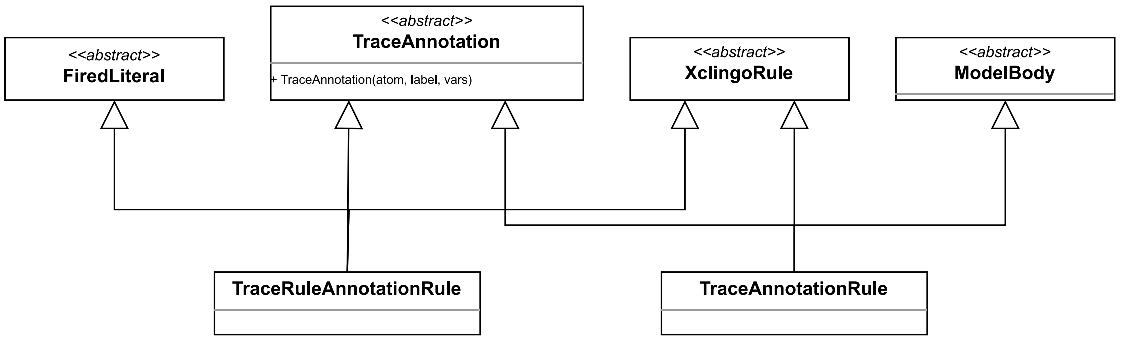


Figure 4.8: Decorated components for *trace* annotation rules.

4.4.2 Explaining module

The goal of the *Explaining* module is to solve the explainer program and provide the explanations to the user. The design of this module is critical for the Python API user experience and for the maintenance of the `xclingo`'s CLI tool, which acts as a user for this module. In this sense, the design of this module tries to *extend* `clingo`'s Python API, providing it with the ability to provide explanations but without modifying its interface. Figure 4.9 depicts the design of the main classes of the module. From the Python API user's perspective, everything is masked behind the `XclingoModel` class method

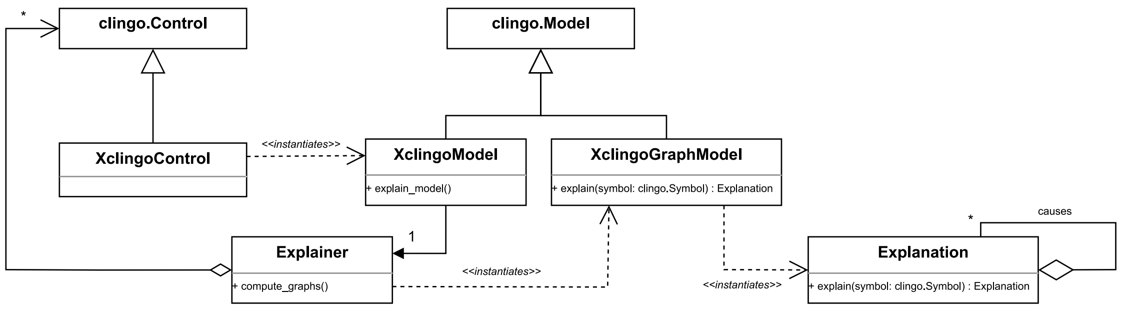


Figure 4.9: Design for the main classes of the *Explaining* module.

`compute_graphs`. The classes `XclingoControl` and `XclingoModel` directly inherit from `clingo`'s `Control` and `Model` classes respectively so that the use apart from obtaining explanations is the same with `clingo`'s and `xclingo`'s API. When such a method is invoked, the explainer program is built and solved, but everything stays transparent to the user which just receives an iterator over a sequence of `XclingoGraph-Model` objects. These objects represent the answer set of the explainer program (i.e. the support graphs). As their corresponding class also inherits from `clingo`'s `Model` class, they can be inspected as usual to retrieve the atoms representing the graphs or `Explanation` Python objects can be obtained through the `explain`*symbol* method.

4.4.3 Extensions

Having an important part of the functionality written as an ASP specification means that we can leverage the advantages of a declarative paradigm. Beyond the implications for the maintenance of the code itself, it should be now very easy for an ASP engineer to extend or modify crucial parts of the default behavior of `xclingo`. Of course, this implies being familiar with the particular specification, which is fully explained in Section 4.3. This can be used for a wide variety of goals, from adapting the output to other system's input to reasoning about the explanations themselves.

In fact, `xclingo` already makes use of this idea to implement some of its base features. A good example of this is the *auto-tracing*. Recall this feature already explained in 4.2.3 which is used to generate automatic traces for the atoms in the explanations. Once we have an support graph, encoded with the predicates used by the tool, adding automatic traces for every atom (i.e. node) in the graph is as easy as writing the following rule.

```
% Genrerates a label for each atom in the explanation
_xclingo_label(Atom, Atom) :- _xclingo_f_atom(Atom).
```

In this way, each `Atom` included in the explanation will have an associated trace that looks exactly like the atom. This rule is saved in a file called *autotrace_all.lp*, which is used as an extension whenever the user uses the option `-auto-tracing=all`. If the user uses the option `-auto-tracing=all`, the extension *autotrace_facts.lp* is used instead. The rule used in the latter case is shown below.

```
% Genrerates a label for each fact atom in the explanation
_xclingo_label(Atom, Atom) :- _xclingo_f(_, _, Atom, ()).
```

As the `_xclingo_f` predicate fourth argument is a tuple containing the atoms from the body of the rule, we can assume the atom is a fact when the tuple is empty. The same idea can be used to generate labels only for particular atoms, perhaps based on the causal information gained from the graphs like labeling the atoms that are caused by one another.

Another interesting example of this is to compute the indirect causes of the atoms. Given a support graph, the original encoding of `xclingo` already provides the edges of the graph as the direct causes. The edges of the support graph in several phases can be queried: the original support graph, the graph after edge running and the graph after node forgetting. For instance, to obtain the indirect causes of an atom in the original support graph trivially follows from the code below:

```
indirect_cause(Effect, Cause) :- direct_cause(RuleID, Effect, Cause).
indirect_cause(Effect, Cause) :- indirect_cause(Effect, Y), direct_cause(Y, Cause). % transitivity
```

This can also be used to form complex queries over the support graphs. For instance, one could obtain only explanations where a particular atom is indirectly caused by another. The extension code below will restrict the explanations graphs only to those where Gabriel is in prison because he resisted authority

```
:- not indirect_cause(sentence(gabriel, prison), resist(gabriel)).
```

As we are explaining, the extensions can be used to modify as well as to extend the behavior of `xclingo`. The use of pure ASP for the extensions provides them with outstanding flexibility. When using extensions one can see `xclingo` as a framework rather than an explanation tool. In such a framework, `xclingo` computes the support graphs for the model concerning a program and lets the user write some code that leverages the graphs in some way. Indeed, `xclingo` just interprets the input ASP program as causal, but it is agnostic to the aim the programmer has. A very naive example to illustrate this would be building a Hamiltonian path checker using `xclingo`. The rules themselves would be used to represent the edges of the graph. For instance, if node `a` has incoming edges from `b` and `c` we could write the rule

```
a :- b,c.
```

By doing this with the rest of the nodes, `xclingo` will internally draw the graph, and now we can include our Hamiltonian checker as an extension.

Chapter 5

Commonsense Explanations with `xclingo`

As stated in the introduction, the main goal of this dissertation is to provide ASP systems with commonsense explanations. After introducing the tool `xclingo` which computes support graphs and further allows the user to *design* the final explanations obtained, in this Chapter we give notions on how to use those features to actually get commonsense explanations. We start by contrasting the differences between technical and commonsense explanations and further discuss their importance and the role they can play in making ASP systems both transparent and accountable.

In Section 5.2 we make an important observation about the explanations that we can obtain from strong equivalent programs. In particular, two strong equivalent ASP programs can produce different explanations. This has important consequences that are discussed throughout the chapter, such as that explanations from efficient ASP encodings may not be suitable for commonsense explanations even if they obtain the same solutions as the original encodings. In the line of demonstrating this, we provide a practical example in Section 5.3 and a practical solution to the problem as well in Section 5.4.

Another topic of great importance that was introduced at the beginning of this dissertation is that most of the explanations requested by humans are not positive, *"How come p to be true?"* but rather most complex causal queries such as contrastive questions that require counterfactual answers. Section 5.5 further discusses these topics and proposes the design of a system that is able to answer this type of causal query, including answers to *Why not?* questions. A practical example of how to implement this system using `xclingo` is provided.

5.1 Technical vs Commonsense explanations

Many AI projects fail when reaching the deployment phase, after all the work to obtain systems that demonstrate to be able to solve the task they were modeled or trained for. Especially in critical domains such as medicine or law, the main problem these projects face is how to gain the user's trust. Even when the performance of the system in the task is statistically sound, situations where the expert and the system do not agree generate important trust issues that are hard to solve. Of course, this may happen between two different human experts too, but the important difference in that case is that they can argue about their opinions, reach conclusions and ultimately get to know which decision they should trust. Even if they do not agree, each expert does not necessarily decrease the trust in each other, if the arguments that justify each decision make sense. This kind of situation would be desirable with automatic decision systems too.

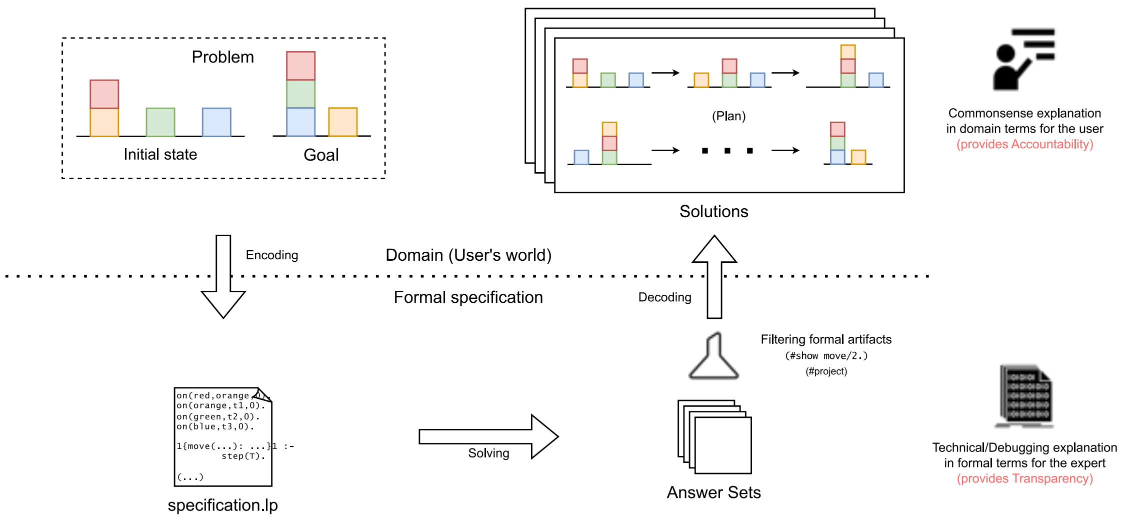
In the case of machine learning, the most widely used algorithms produce models that work as *black boxes* in the sense that a user cannot understand why the model makes a decision. Even in the cases when the models are readable, as with decision trees, they can be hard to understand if they are too large, or have too many features. And even when, apart from readable, they are small and easy to follow, that does not mean that the user will trust the explanations you can extract from them. For instance, the conditions that the input instance has to meet until reaching a leaf node in a decision tree, although very effective as a classification test, may make no sense for a healthcare professional. Something similar happens with other symbolic approaches. Sure, one can check the ASP rules from the program to see why exactly an atom is being derived. But with very few exceptions, in most cases, the final user may have serious difficulties following that derivation, substantially reducing her trust in the system. According to Dignum [34], the property of *transparency* refers to the capability to describe, inspect and reproduce the mechanisms behind the system decisions and the provenance and dynamics of the data that is used or created by the system. Systems like decision trees or ASP programs are transparent since one can follow how the final decision is made from input to output. Yet, this claim is not fully accomplished by ASP solvers in the sense that some parts of their behavior are not easily predictable. For instance, the ordering in which multiple answer sets are generated or the time to obtain one or more solutions may drastically depend on the configuration of the multiple heuristics involved and/or on the optimizations implemented in the solver.

Accountability on the other hand, is the property that a system has when can justify or explain its conclusions to users and other relevant actors. In the same line we were emphasizing before though, this has to be done in a way that each actor understands the reasoning behind the explanation for increasing the trust in the system.

In the later years, a lot of approaches for the explainability of AI systems were proposed, each of them proposing a different concept of what an explanation is. In the case of ML models, for instance, novel approaches like LIME [88] or SHAP [68], both model agnostic, disagree on what is an explanation in the first place. For the former, an explanation is a linear model learned from several generated instances around a particular prediction. Very differently, for the latter, an explanation is an estimation of the contribution of each input feature to the conclusion. In the world of logic programming, the situation is at least a little more homogeneous. In general, the agreement seems to be that an explanation for logic programs has to refer to the derivation of the atoms themselves in some way. The survey [48], published in 2019, visits and compares some of the *state-of-the-art* approaches to explainability in ASP. In Chapter 6, we also review some of these approaches, including some new approaches that have been published after the survey. For instance, systems like xASP [4, 99] (currently in its second version xASP2), or s$CASP$ [7] use *Directed Acyclic Graphs* and *Justification Trees* (respectively) to explain Answer Set Programs. The differences between approaches are based on other aspects like whether the default negation is included or not in the explanations, which aspect from ASP language are supported, or if the explanations are computed from ground atoms or in a top-down manner, among others. However, disregarding whether they focus on machine learning or logic programs, most of these approaches only provide *transparency* to the systems. Furthermore, most of them rely solely on technical concepts, that a final user is normally not familiar with. In the case of logic programs, a final user would need to have an advanced understanding of logic programming to fully understand the explanations. But if the aim is to provide accountable explanations, any approach should worry more about providing something that speaks in a non-technical language and that makes sense for the non-technical users.

In any software development, there is a clear difference between the technical and the user's world. This is, the user stands her requirements in natural language, as an abstract set of features or goals, and it is the responsibility of the development team to translate those requirements and to design some technical implementation that supports such requirements. It would be considered a mistake, however, to present the user with an interface that makes use of concepts outside of the user's world. Say for instance that a user asks to search over the set of products, and the software shows a JSON object containing the list of the search results as a response. Or that the user asks to know the distribution of the sales of the last month and the software shows the list of all the sales together with their corresponding dates. Sure, the information that the user has requested is there, but the software is neither facilitating the interpretation nor speaking in the user's terms. Much more adequate would be to show the products in a (probably smartly ordered) table and display a fancy histogram depicting the distribution of the sales. Speaking the user's language is a crucial piece when building software that aims to satisfy non-technical user needs, often having a greater impact on the deployability of the tool than the technical excellence or even the performance.

The same dilemma happens when we face solving with KR. Normally, the users are not directly provided with the raw atoms from the answer sets of the program, but a fancy interface that collects that output and presents it adequately is developed. Figure 5.1 depicts the typical KR pipeline and emphasizes how the different steps belong either to the domain's or user's world in the upper half or to the formal specification scope. We typically start with a description of the problem in user's terms,



```
on(red,orange,0).
on(orange,t1,0).
on(green,t2,0).
on(blue,t3,0).

1{move(...): ...}1 :-
          step(T).

(...)
```

Figure 5.1: Explanations in the context of the KR workflow.

usually in natural language. The main task of the KR engineer is to encode such a description as a set of rules in a logic program. This set of rules can be easier or harder to understand but, in the end, it lies in the Formal specification world as the ASP code is itself a technical artifact. This encoding is then solved and the solutions are obtained in the form of sets of atoms (i.e. answer sets). Unfortunately, from the point of view of most final users, these atoms are still a cumbersome technical artifact that belongs in the technical (formal specification) world. In the same way, displaying an explanation that is conformed only with the atoms from the answer sets still is a technical artifact that should not be shown to a final user. This kind of *technical explanation* provides *transparency* to the system but it is still far from providing *accountability* nor being what we call a *commonsense explanation*. Among all the atoms within the answer sets, it is just a subset of them that represents the solution. From this fact, additional filtering usually follows after obtaining the answer sets. In the case of `clingo` solver, this involves the use of `#show` sentences to only display the atoms that are considered relevant for representing the solution. At this point, an extra effort can be made to design the displayed predicates, hiding the complex KR representation behind them and showing the solution in the clearest way possible. What can drive us back to the user world is to use additional software to decode and represent the solutions in the user's terms. This of course implies making an extra development effort, that requires a different type of knowledge than the typical KR engineering. To that aim, it also exists ASP based software to integrate both the encoding with the solution displaying such as `clingraph`[1] but any ad hoc solution could work. In parallel, we need to make an extra effort when designing the explanations for them to meet the user's requirements. In the case of `xclingo`, this effort is firstly directed towards annotating the program.

For setting up the terminology, we refer to the term *Technical Explanation* as defined in Definition 2 in Chapter 1. Such an explanation speaks in technical terms and provides transparency to the system. They are useful for debugging and testing the system, proving its correctness or even checking some nonfunctional requirements. Thus, the type of user this kind of explanation is directed to is an expert in the software the system is built in, in our case ASP. For that reason, the language the explanation needs to speak is a technical one. For instance, showing the complete derivation proof of an atom by using the `-auto-tracing=all` option would be an example of a *technical explanation*. In the case of ASP, we further have different levels of technical explanations. The technical explanation may be at the level of the inference mechanism according to the answer set semantics (thus, being independent of the solver). This is the case of *Support Graphs*, `xclingo` and most of the ASP explainability approaches. But we could also provide explanations according to the specific solver behavior, following its execution steps, heuristics, optimizations, etc.

On the other hand, we refer to Definition 1 also in Chapter 1 for the term *Commonsense Explanation*. Such is an explanation that argues or justifies the system's conclusion as a (selected) causal, real-world explanation that speaks in a language the final user is familiar with. However, although it is one of the most common cases, this does not always imply that natural language is the best way to prompt the explanation. Sometimes, a diagram, a graph or even a chart can be the best support for the explanation that the user needs. It is also of great importance to note that an explanation of this kind does not necessarily reflect the internal computation process the system is doing to reach the conclusion. This process often does not match the real causal reasoning humans use to explain the events of the real world that the system is talking about. Moreover, an intelligent implementation will usually make simplifications that break this causal reasoning but enhance the performance of the system. However, if we are aware of the simplifications we can still provide a solution to get the explanations back in the causal framework the user is expecting it to be.

Indeed, the *encoding* step from Figure 5.1 can often be divided in two as in Figure 5.2. First, the engineer encodes the problem in the most straightforward ASP rules she can come up with. That spec-

[1]https://github.com/potassco/clingraph

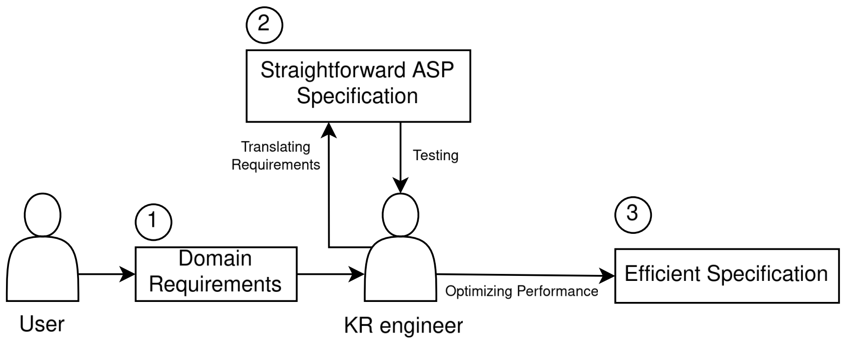


Figure 5.2: Steps for encoding the user requirements as an ASP program.

ification is then tested to check that it meets the requirements. This first specification is not usually the best in terms of performance but is the most straightforward encoding of the natural language requirements and thus, its correctness is typically easy to check. As a result is often a good causal representation from which `xclingo` can obtain good explanations. After that, it is iteratively modified to an equivalent version that increments the efficiency, often at the cost of losing the comprehensibility of the ASP code and requiring a harder effort for the verification of its correctness. In the case of `xclingo`'s explanations for ASP programs, the explanation graphs are directly tied to the particular rules that form the ASP program. In that sense, any simplification that we can make during computation will affect the space of explanations that we will be able to obtain through `xclingo`, even if the original and the simplified encodings are equivalent. This means that, as we demonstrate in the next section, sometimes the markup annotation language (although significantly flexible) can be insufficient to design commonsense explanations that meet the user's requirements.

5.2 Strong equivalence does not suffice

In the context of Answer Set Programming, two programs P_1 and P_2 are said to be *strongly equivalent* [66] if for any other program P_3, it holds that $P_1 \cup P_3$ has the same answer sets than $P_2 \cup P_3$. This is a well-known property in the field that was born to simplify parts of a logic program without looking at the rest of it and is a stronger version of equivalence as the mere coincidence in the set of answer sets.

Figure 5.3 depicts two circuits from an example in [80]. Note how both circuits behave the same for any input for the switches $sw1$ and $sw2$.

Figure 5.4 contains the ASP specification for both circuits, which are strongly equivalent. Then, Figure 5.5 shows some common code for both circuits, where we add a choice rule for the switches and some traces to obtain explanations.

By solving both circuits, we obtain a total of 4 answer sets, listed in Figure 5.6, corresponding to each possible combination of states (up, down) for the two switches. As both programs are (strongly) equivalent, the state of light is the same in all 4 cases, disregarding the circuit.

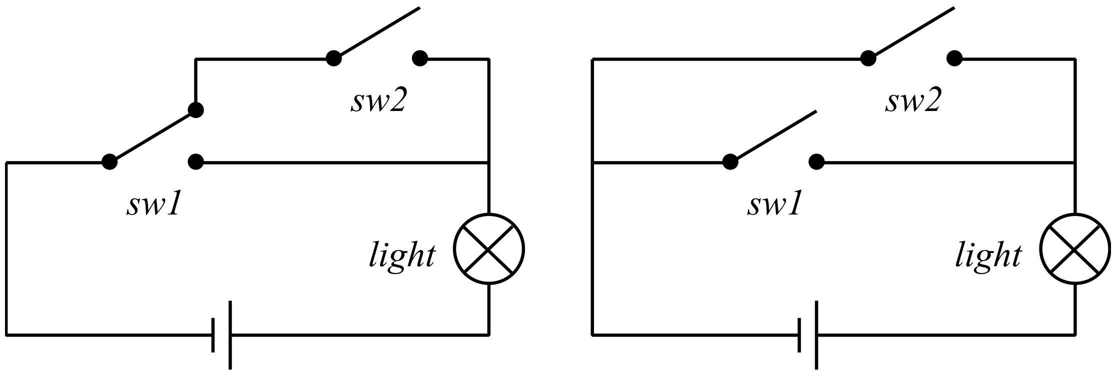


Figure 5.3: Example from [80]. It depicts *Circuit 1* (left side) and *Circuit 2* (right side).

```
% Circuit 1
light(on)  :- sw(1,down).
light(on)  :- sw(1,up),sw(2,down).
light(off) :- sw(1,up), sw(2,up).
```

```
% Circuit 2
light(on)  :- sw(1,down).
light(on)  :- sw(2,down).
light(off) :- sw(1,up), sw(2,up).
```

Figure 5.4: ASP specification corresponding with *Circuit 1* and *Circuit 2* from Figure 5.3.

```
% Adds switches and traces
switch(1;2).
1 {sw(X,up);sw(X,down)} 1 :- switch(X).
#show light/1.
#show sw/2.

%!trace {sw(X,V), "switch % is %",X,V} :- sw(X,V).
%!trace {light(V), "the light is %",V} :- light(V).
%!show_trace {light(V)}.
%!show_trace{sw(X,Y)}.
```

Figure 5.5: Common ASP code for Circuit 1 and Circuit 2.

```
Answer: 1
sw(1,down) sw(2,down) light(on)
Answer: 2
sw(1,down) sw(2,up)   light(on)
Answer: 3
sw(1,up)   sw(2,up)   light(off)
Answer: 4
sw(1,up)   sw(2,down) light(on)
SATISFIABLE
```

Figure 5.6: Answer sets for both circuits 1 and 2 from Figure 5.4 when solved together with the common code from Figure 5.5.

However, it can be easily seen just from the representation of each circuit, how the reasons for the state of the light are different. Thus, it is not surprising that the explanations obtained by `xclingo`, shown in Figure 5.7, reflect these differences.

```
Answer: 1  {sw(1,down) sw(2,down)}
##Explanation: 1.1
  *
  |__"the light is on"
  |  |__"switch 1 is down"







##Total Explanations:   1

Answer: 2  {sw(1,down) sw(2,up)}
##Explanation: 2.1
  *
  |__"the light is on"
  |  |__"switch 1 is down"

##Total Explanations:   1

Answer: 3  {sw(1,up)   sw(2,up)}
##Explanation: 3.1
  *
  |__"the light is off"
  |  |__"switch 1 is up"
  |  |__"switch 2 is up"

##Total Explanations:   1

Answer: 4  {sw(1,up)   sw(2,down)}
##Explanation: 4.1
  *
  |__"the light is on"
  |  |__"switch 1 is up"
  |  |__"switch 2 is down"

##Total Explanations:   1
Models: 4
```

```
Answer: 1  {sw(1,down) sw(2,down)}
##Explanation: 1.1
  *
  |__"the light is on"
  |  |__"switch 1 is down"

##Explanation: 1.2
  *
  |__"the light is on"
  |  |__"switch 2 is down"

##Total Explanations:   2

Answer: 2   {sw(1,down) sw(2,up)}
##Explanation: 2.1
  *
  |__"the light is on"
  |  |__"switch 1 is down"

  ##Total Explanations:   1

Answer: 3  {sw(1,up) sw(2,up)}
##Explanation: 3.1
  *
  |__"the light is off"
  |  |__"switch 1 is up"
  |  |__"switch 2 is up"

##Total Explanations:   1

Answer: 4  {sw(1,down) sw(2,up)}
##Explanation: 4.1
  *
  |__"the light is on"
  |  |__"switch 2 is down"


##Total Explanations:   1
Models: 4
```

Figure 5.7: Explanations for both circuits 1 (left) and 2 (right) from Figure 5.4 when solved together with the common code from Figure 5.5.

In the figure, we have included the state of each switch along with each answer set for enhancing the readability, but this is not a feature of `xclingo` yet at this point. If we look at Answer 1, where both switches 1 and 2 are down and the light is on, the only cause in Circuit 1 is switch 1, while in the case of Circuit 2, each switch constitutes an alternative cause. In the case of answers 2 and 3, the explanations for both circuits are the same. However, in the case of Answer 4, where switch 1 is up and switch 2 is down, both switches are necessary to explain the light being on in Circuit 1, whereas only switch 2 is a cause for Circuit 2.

This small example demonstrates how *strong equivalence* is not a sufficient condition to produce the same explanation graphs. As a result, this complicates the workflow described in Figures 5.1 and 5.2

from Section 5.1. In particular, when we optimize the encoding after being happy with the straightforward one, we usually try to obtain something equivalent but faster. However, we cannot rely on the explanations obtained from both encodings to be the same anymore, even when they are *strongly equivalent*. Furthermore, in most cases, we will probably deal with a tradeoff between performance and obtaining adequate explanations. In other words, we should now consider a new step in the KR workflow that we could call explanation design, which represents an extra effort that we make as KR engineers to design a representation that provides adequate explanations to the user. In the sense that they are causally aligned with the problem they represent (like the real circuits in the figures). This step involves effort in modifying the specification, but only after understanding the user needs to identify the requirements tied to the explanations.

5.3 A Practical Example of Explanation Design: Blocks World

To illustrate the problems we can face when introducing explanation design into the KR workflow, let us use the *Blocks World* domain as a running example. We will visit different specifications for solving the problem that, although equivalent, produce different explanations.

The Blocks World problem is a classical puzzle in the AI Planning field. In this problem, we consider several blocks which can be placed on the table or on top of other blocks. The goal is to transform a given initial arrangement of blocks into a desired configuration or order. Figure 5.8 shows a typical example of a blocks world problem setup.

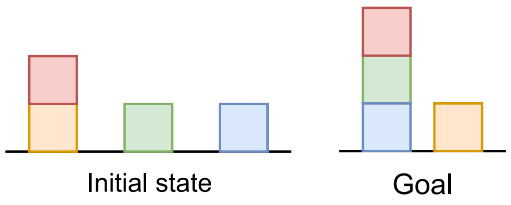


Figure 5.8: Typical set up of a blocks world problem.

Under this representation, a solution consists of a sequence of actions, each one associated to a timestep, where one block is moved from one stack to another. The main constraints regarding these actions are that only the block on top of each stack can be moved, and the selected block can only be placed on the block currently at the top of another stack. This problem is a well-known example in the study of automated planning and is widely used to test and evaluate planning algorithms and knowledge representation formalisms.

Program 5.1 shows the specification of an instance of the Blocks World problem.

Predicate `h(Block, Location, Time)` (namely *holds*) is used to describe the position of the blocks at each particular timestep. Note that, since we have to describe the initial state, we are representing the position of each block at timestep `0`. Predicate `g(Block, Location)` (namely *goal*) represents the goal state. That is, the position where we must find each block at the end of the plan.

Program 5.2 contains some common code for the four encodings that we will be visiting.

Lines 1 and 2 contain some domain predicates such as the `blocks` or the `locations`. Lines from 4 to 7 represent the executability constraints that ensure that the action's preconditions are satisfied. We

```
#const last=3.
#const n=4.
h(on(2),table,0). h(on(3),table,0).
h(on(1),3,0).     h(on(4),1,0).
g(on(4),table).   g(on(1),4).
g(on(2),1).       g(on(3),table).
```

Program 5.1: ASP specification of an initial state for the Blocks World problem.

```
time(0..last). step(1..last). block(1..n).
location(table). location(B):-block(B).

unclear(C,T) :- h(on(B),C,T),C!=table,time(T).
:- o(move(B,_),T), unclear(B,T-1).                  % executability
:- o(move(_,L),T), unclear(L,T-1).                  % executability
:- o(move(B,table),T), h(on(B),table,T-1), step(T).  % executability
:- g(on(B),L), not h(on(B),L,last).

h(F,V,T) :- h(F,V,T-1), not c(F,T), step(T).        % inertia
c(F,T)   :- h(F,V,T-1),h(F,W,T),V!=W, step(T).      % changes

% Text labels
timetext(0,initially). timetext(last,finally).
%!trace {h(on(B),L,T), "Block % is % on %",B,Txt,L} :- h(on(B),L,T), timetext(T,Txt).
%!trace {h(on(B),L,T), "Block % is now on % at %",B,L,T} :- h(on(B),L,T), h(on(B),L',T-1), L!=L', T!=last,T!=0.

%!show_trace {h(on(B),V,last)}.
```

Program 5.2: Common annotated ASP code for the Blocks World problem.

find the constraint for the goal state in Line 8. The rule in line 10 models inertia (a fluent F preserves its previous value V if F has not changed, $c(F, V)$) and, the rule in line 11 models change (when the inertia is broken). Change is represented with predicate `c(Block, Time)`. From line 13 on, extra code and annotations to write some text traces are added.

Recall now the workflow pipeline from Figure 5.2 and imagine that, after some engineering effort, we come up with the efficient specification in Program 5.3 for modeling the generation of the plan (i.e. the movements of the blocks in each time step). We will call it Encoding 1 from now on.

```
1  action(move(B,table)):- block(B).
2  action(move(B,C)):- block(B),block(C),B!=C.
3
4  1 {o(A,T): action(A) } 1 :- step(T).
5  h(on(B),L,T) :- o(move(B,L),T).
6
7  %!trace {o(move(B,L),T), "We moved block % to % at step %",B,L,T} :- o(move(B,L),T).
```

Program 5.3: Encoding 1.

In Encoding 1, any action can be selected for each time step (line 4) and the state of moved blocks is captured at line 5. The annotation at line 7 labels each movement with a descriptive text. The explanations obtained by `xclingo` for Encoding 1 can be seen in Output 5.1.

```
  *
  |__"Block 14 is finally on 13"
  |  |__"Block 14 is now on 13 at 22"
  |  |  |__"We moved block 14 to 13 at step 22"
```

Output 5.1: Explanations obtained for Encoding 1.

Contrary to what a common user would expect, the explanation for the final position of block 14 does not include the entire history of movements concerning the block, but rather only talks about the final step 22. Let us detailedly explain why this is happening. First, note that the *user* is asking `xclingo` to explain the position of all blocks at the last time step. This is done through the `show_trace` annotation written in line 19 of Program 5.2. We will only show the explanations for block 14 for brevity. Bear in mind that `xclingo` interprets the set of rules in the program as cause-effect relations. If we follow the rules we see that, for the particular Encoding 1 (Program 5.3), `h(F,V,T)` depends on `o(move(B,L,T))` from the body of the rule in line 5 of Program 5.3 which in turn depends on `action(A)` and `step(T)` from rule in line 4. Both last mentioned predicates are domain, factual predicates that do not depend on anything, meaning an end for the cause-effect chain that `xclingo` is unraveling. This can be easily checked by enabling the `auto-tracing=all` option. In this way, we can check all the involved cause-effect relations that `xclingo` is considering. Output 5.2 shows the explanation we get for block 14 when enabling such an option.

`xclingo` cannot generate an explanation for any `h(B,L,T)` atom that shows the history of movements previous to instant `T` since there is no cause-effect connection between one state and its previous ones (neither directly nor transitively). From a pure KR (non-explainability aware) point of view, it makes sense that the choice rule in line 5 of Program 5.3 does not include the previous position of the

```
*
|__"Block 14 is finally on 13";h(on(14),13,23)
|  |__"Block 14 is now on 13 at 22";h(on(14),13,22)
|  |  |__"We moved block 14 to 13 at step 22";o(move(14,13),22)
|  |  |  |__action(move(14,13))
|  |  |  |  |__block(14)
|  |  |  |  |__block(13)
|  |  |  |__step(22)
|  |__step(23)
```

Output 5.2: Explanation for block 14, using Encoding 1 and enabling `auto-tracing=all` option.

block because it is not needed to generate the next movement. Indeed, adding such a condition would slow down the solver when computing plans and have no effect on the solutions obtained. To eventually obtain an explanation that includes the complete history of actions several modifications to the original program may be done. Let us now introduce Encodings 2 and 3 and discuss how they fulfill this goal in different ways.

Encoding 2 (Program 5.4) only introduces one change with respect to Encoding 1.

```
action(move(B,table)):- block(B).
action(move(B,C)):- block(B),block(C),B!=C.

1 {o(A,T): action(A) } 1 :- step(T).
% h(on(B),L,T) :- o(move(B,L),T). % replaced
h(on(B),L,T) :- o(move(B,L),T),  h(on(B),L',T-1).

%!trace {o(move(B,L),T), "We moved block % to % at step %",B,L,T} :- o(move(B,L),T).
```

Program 5.4: Encoding 2.

This change concerns the effect axiom (originally at line 5, Program 5.3), which now includes a new literal recalling the position `L'` of block `B` at the previous state `T-1`. Note how that previous location `L'` is not used at all in the rule head to obtain the current step's position, but just to introduce the transitive cause-effect relation needed to generate the desired explanations. Output 5.3 shows the explanation for block 14 when replacing Encoding 1 with Encoding 2.

```
1     *
2     |__"Block 14 is finally on 13"
3     |  |__"Block 14 is now on 13 at 22"
4     |  |  |__"We moved block 14 to 13 at step 22"
5     |  |  |__"Block 14 is now on table at 16"
6     |  |  |  |__"We moved block 14 to table at step 16"
7     |  |  |  |__"Block 14 is initially on 11"
```

Output 5.3: Explanations obtained for Encoding 2.

Note how *movement* and *state* causes in the explanations are grouped in the same indentation level, showing how they are *joint causes* for the next position of the block. However, if we take lines 3 to 5 as an example we can see how a movement at step 22 and the state achieved at step 16 act together as a

```
*
|__"Block 14 is finally on 13";"Block 14 is not moved from 13 at 23"
|  |__"Block 14 is now on 13 at 22"
|  |  |__"We moved block 14 to 13 at step 22"
|  |  |__"Block 14 is not moved from table at 21"
|  |  |  |__"Block 14 is not moved from table at 20"
|  |  |  |  |__"Block 14 is not moved from table at 19"
|  |  |  |  |  |__"Block 14 is not moved from table at 18"
|  |  |  |  |  |  |__"Block 14 is not moved from table at 17"
|  |  |  |  |  |  |  |__"Block 14 is now on table at 16"
|  |  |  |  |  |  |  |  |__"We moved block 14 to table at step 16"
|  |  |  |  |  |  |  |  |__"Block 14 is not moved from 11 at 15"
|  |  |  |  |  |  |  |  |  |__"Block 14 is not moved from 11 at 14"
|  |  |  |  |  |  |  |  |  |  |__"Block 14 is not moved from 11 at 13"
|  |  |  |  |  |  |  |  |  |  |  |__"Block 14 is not moved from 11 at 12"
|  |  |  |  |  |  |  |  |  |  |  |  |__"Block 14 is not moved from 11 at 11"
|  |  |  |  |  |  |  |  |  |  |  |  |  |__"Block 14 is not moved from 11 at 10"
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"Block 14 is not moved from 11 at 9"
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"Block 14 is not moved from 11 at 8"
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"Block 14 is not moved from 11 at 7"
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"Block 14 is not moved from 11 at 6"
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"Block 14 is not moved from 11 at 5"
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"Block 14 is not moved from 11 at 4"
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"Block 14 is not moved from 11 at 3"
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"Block 14 is not moved from 11 at 2"
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"Block 14 is not moved from 11 at 1"
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"Block 14 is initially on 11"
```

Output 5.4: The effect of tracing the intertia rule.

joint cause even though the rule from Program 5.4 explicitly recalls time step `T-1` This is the effect of *forgetting* non-traced atoms in the explanation.

To understand this, let us recall the different traces affecting this situation. `h(on(B),L,T)` atoms are traced at the initial and final step (see line 16 of Program 5.2), and when the corresponding state of the blocks changes with respect to its previous state (see line 17 of Progra 5.2m). Finally, one last annotation *traces* any generated movement of a block `o(A, T)`, but this annotation is defined together with the predicate at the end of each encoding (programs 5.3 and 5.4). These three annotations generate all the text included in the final explanations.

The cases for the state of block 14 in step 22 (line 3 of Output 5.3) are the generated movement together with the its previous position However, this previous state of block 14 did not receive any text, since this only happens the instant the state of a block changes. The most recent change in the state of block 14, as we can read in the explanation, occurs at step 16, when we move it from the top of block 11 to the table. The `h/3` atoms from step 17 to step 21 were forgotten from the graph since they were not *traced*. As a result, actions on steps 22 and 16 act as causes for the new position, which can be seen as misleading at first.

One straightforward idea that we could have to solve this, is to add a text to all the states at any time step, obtaining the full chain of stages of any block. This can be easily obtained by *tracing* the *intertia* rule at line 10 in Program 5.2. However, the result is almost unreadable, showing too much *irrelevant* information, as we can see in Output 5.4.

In a temporal setting, users will be interested in when and how the fluents *change* rather than knowing the value for the fluent in every time step of the plan. Instead of having a large explanation repeating irrelevant information for every time step, it is much more interesting to only highlight the changes and to show the intervals in which the fluent did not change. Once we have seen why tracing inertia is not a good solution for the problem of Encoding 2, let us introduce how Encoding 3 solves the issue. This encoding introduces the same dependence that Encoding 2 but this time in the choice rule instead (see Program 5.5), leaving the rest of the code untouched concerning Encoding 1.

```
action(move(B,table)):- block(B).
action(move(B,C)):- block(B),block(C),B!=C.

% 1 {o(A,T): action(A) } 1 :- step(T). % replaced
1{o(A,T): action(A), A=move(B,L), @h(on(B),L’,T-1)@}1 :- step(T).
h(on(B),L,T) :- o(move(B,L),T).

%!trace {o(move(B,L),T), "We moved block % to % at step %",B,L,T} :- o(move(B,L),T).
```

Program 5.5: Encoding 3.

Similarly to Encoding 2, the previous position is introduced as a cause in the condition of the conditional atom of the choice rule but it is not used for obtaining the new movement. Therefore, in the obtained explanations shown in Output 5.5, we can observe something similar to what was achieved with Encoding 2, but this time every consequent cause is in a new level of indentation.

```
 *
 |__"Block 14 is finally on 13"
 |  |__"Block 14 is now on 13 at 22"
 |  |  |__"We moved block 14 to 13 at step 22"
 |  |  |  |__"Block 14 is now on table at 13"
 |  |  |  |  |__"We moved block 14 to table at step 13"
 |  |  |  |  |  |__"Block 14 is initially on 11"
```

Output 5.5: Explanations obtained for Encoding 3.

With Encoding 3, the structure of the explanations is finally closer to what a common user would find intuitive. However, certain literals have been added to the program only to model the causal relationships and not to solve the problem, which feels unclean and also may introduce additional computing and affect performance. Table 5.1 shows how different encodings affect solving and explaining time. For each Encoding, solving time, explaining time and the addition of both are shown in seconds. Among each time measure, the percentage that the sample represents of the total is shown between parentheses.

	Encoding 1	Encoding 2	Encoding 3
Solving Time	1.187s (45%)	3.503s (83%)	3.614s (61%)
Explaining Time	1.356s (55%)	0.706s (17%)	2.274s (39%)
Total	2.453s	4.209s	5.888s

Table 5.1: Times for different encodings.

From these results, it is possible to see the tradeoff between code optimization and the *explanation design*. The existence of any "good practices" that achieve good explanations and at the same time fast specifications have yet to be investigated and (if they exist) are likely to be highly dependent on the type of problem and domain. However, even if we cannot reconcile explanation design and speed in a single specification, we can still combine the two using some tricks with `xclingo`.

5.4 Model Feeding for Fast Commonsense Explanations

There is a technique we can apply with `xclingo` to leverage the best of both worlds: fancy, commonsense explanations together with fast solving. The workflow remains the same: first, we work on a straightforward specification of the problem that is replaced (after some enhancements) by a faster specification. We will refer to this latter specification as *fast encoding*.

Then, we invest additional effort in explanation design, modifying the fast encoding until we achieve a (very likely slower) version of the specification that meets the user requirements in terms of explanations. This will be called the *explaining encoding*. Now, the idea is to use the *fast encoding* for obtaining the models, to later explain them using the *explaining encoding*. We can do this using the `-feed-model=<file>` option from `xclingo`. The file expected by the option should contain a valid model obtained using the *fast encoding* and any solver. We refer to this technique as *model feeding*. Figure 5.9 shows the model feeding workflow while Figure 5.10 shows how the data flows when applying the approach.

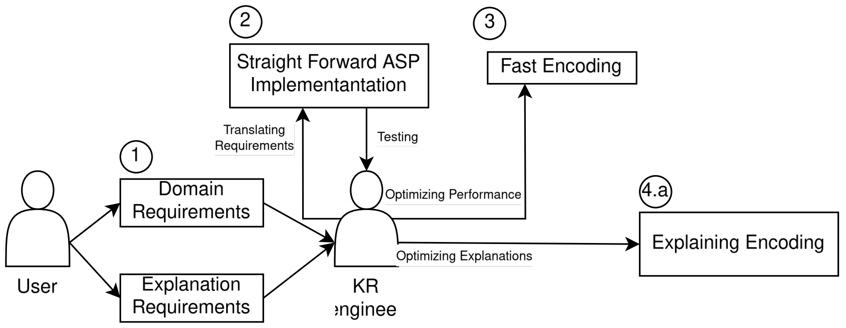


Figure 5.9: KR workflow from Figure 5.2 updated to fit the *model feeding* approach.

Using this approach we can save much time when the solving time of the explaining encoding is much greater than the fast encoding. In the case of the Blocks World example from Section 5.3 we can obtain the explanations from Encoding 3, with the solving time from Encoding 1. For this example, this drastically reduces the time needed to obtain the explanations by 41%.

5.5 Answering Different Types of Causal Queries

5.5.1 Classifying types of answers

In a real-world scenario, explanation queries from the user will usually be natural language questions such as *"Why p?"*, *"Why not q?"*, *How come s?* or *"Why is there no solution?"*. As they are stated in

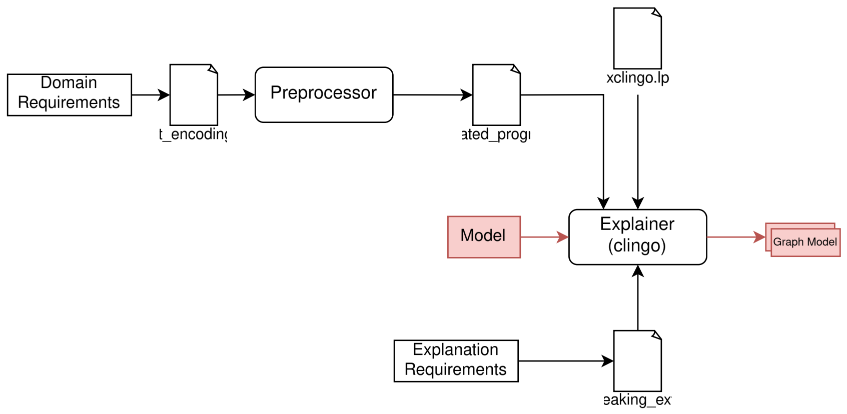


Figure 5.10: Data flow when applying model feeding approach.

natural language, it is natural for these questions to be ambiguous and, as discussed in Chapter 1, to include implicit information. Thus, it is normal that different ASP explainability approaches end up producing different answers for the same questions.

For instance, for the question *"How come p?"*, *Causal Values* provided by [49] are different from *And-Or Graphs* of [45]. Although both are semantically related, they are different in form, the former an algebraic expression, and the latter a labeled directed graph. Also, different natural language questions can sometimes be answered very similarly. Take, for instance, the questions *"What if p?"* and *"How to get p?"*. Although not the same, both questions implicitly suggest that p is not true in a current solution M. Thus, to implement a valid answer, both questions imply reasoning about an alternative solution and then using it to explain the result.

As Miller shows in [75], sometimes some natural language questions have some additional implicit purpose not mentioned in the query. For instance, he argues that most of the *"Why p?"* questions are implicitly contrastive *"Why p instead of q?"* questions. Therefore, it seems reasonable to explicitly separate the notions of (natural language) *question* and *answer*. In this Section, we propose different abstract answer types and we provide general definitions for them. Also, we propose the kinds of questions that they can solve.

Proof-based explanation

Throughout this dissertation, we have seen how `xclingo`, by default, is able to answer, say *positive*, *"How come p?"* questions. First, once a model M is found `xclingo` can compute the support graphs $G_1, G_2, ..., G_n$ corresponding to M (see Definition 6). Such graphs G_i can be considered explanations for the whole model. Then, to explain *"How come p?"*, we can obtain an explanation for p (following Definition 8) from any G_i for the atom p such that $p \in M$. From now on we will refer to this kind of answer as a *proof-based explanation*.

Consider this term as an abstraction embracing not only our notion of explanation but also any other definition that is built from some kind of derivation starting from a program and one of their models. In fact, in Chapter 6 we will see how most of the approaches to explainability in ASP [48] provide definitions that fall inside (or at least can relate to) this abstraction. Therefore, we define this abstraction as in Definition 12.

Definition 12 (Proof-based Explanation). *A Proof-based Explanation for a query Q with respect to a program P and a model M of P such that $Q \in M$ (or $Q \subseteq M$ if Q is a set of atoms), is an object representing or that can relate to a particular (or several) derivation proof(s) of p using rules in P with respect to M. We denote this as why(Q, M, P).*

Unsatisfiability Explanation

Another type of response of interest that has been investigated recently is what we could refer to as the abstraction *Unsatisfiability Explanation*, defined in Section 13. This is (again, abstractly) the problem of finding the reasons why a program has no models. For *implementing* this abstraction several approaches could be taken.

Definition 13 (Unsatisfiability Explanation). *An Unsatisfiability Explanation of a program P such that $SM(P) = \emptyset$ is an object representing the reasons why P has no model. We denote this as whyunsat(P).*

Note that an unsatisfiability explanation would be answering different natural language questions like *"Why is there no solution?"* but also *"Why is there no solution such that Q"*. This indicates that the same answer type can answer several natural language questions. Moreover, it could be argued how this type of answer could be a valid answer for a question such that *"Why not p?"* in the case that $p \notin M_i$ for all M_i of the program.

How-come Explanations for Why-Not Questions

Let us discuss now answers for natural language questions such as *"Why not Q?"*. First, asking such a question suggests that we are considering a model M such that Q is false in M. Then, recall the first finding by Miller [75], *explanations are contrastive*, that was discussed in Chapter 1. In short, this means that, in the real world, most questions like *"Why p?"* are actually (implicitly) questions like *"Why p instead of q?"*, where q is something that is false but which the *questioner* expects to be true. This suggests the existence of a *default world* where Q is true, that is expected to exist by the *questioner* and that is different from the *actual world* where Q is false. In Miller's [75], the *default world* is referred to as the *fact* whereas the *actual world* is referred to as the *foil*. Moreover, he also points out that meaningful explanations are built on the differences between the *fact* and the *foil*. This connects with Reiter's theory for diagnosis [87] where a similar notion is considered in which the differences between both are called *discrepancies*. Later approaches to diagnosis [11] follow this idea.

Now relating this to explainability in ASP, it seems reasonable to assume that, when a user asks something like *"Why not q?"* is because Q is false concerning the *actual model* M and the user assumes the existence of an *alternative model* M' such that Q is true in M'. Therefore, by providing M' or the differences between M and M' we could answer the user's question. Moreover, a proof-like explanation could be obtained from any of them. This would be a different type of (abstract) answer involving finding the alternative model M' where some query holds. Also, note that such an explanation would also answer also questions like *"What has to change so that p?"*, reinforcing the idea that the same type of answer can solve several natural language questions. Then, we define the abstract answer type *How-To Explanation* in Definition 14, that starting from a model M and a program P such that a particular *foil* Q does not hold, it provides a set of changes in M and/or P such that there exists a model M' where Q holds.

Definition 14 (How To Explanation)**.** *Given a formula Q, a program P and a model* $M \in SM(P)$ *such that* $M \not\models Q$*, a program* P' *such that is obtained by adding or removing rules from P, and an alternative model* $M' \neq M$ *of* P' *such that* $M' \models Q$*, a* How-To Explanation *is either* M'*, or the differences between* $M' and M$*, or an object representing the reasons for q being true in* M' *using rules in* P'*. We denote this as howto*(Q, M, P)*.*

Note that the alternative model M' can still be a model of the original program P when $P' = P$ so that any rule is neither added nor removed.

5.5.2 Answering Causal Questions

In this section, we propose an abstract methodology for answering a subset of explaining questions. For its design, we refer to the different abstract answer types defined in the previous section. A diagram depicting such methodology is shown in Figure 5.11. We will discuss the interpretation of each diagram's branch (from A to F) and how they can be implemented using support graphs and `xclingo`. In the figure, green squares with corner edges represent solving steps that could be performed with any ASP solver, blue squares with rounded edges represent one of the abstract explanation answers defined in the previous section, grey squares with corner edges represent some abstract process implementing either a solving step or an abstract explanation answer, white diamonds represent flow splits depending on some conditions, and finally, white circles represent the end of the flow.

We start from a fixed program P and a fixed model M of P. Then, after inspecting the model, the user poses a question. This should be a question in natural language that would be translated by a system into a query Q that can either be a set of atoms or a formula. We do not provide any particular mapping between questions and queries or answers, although Figure 5.11 includes some suggestions through the different flows.

If Q is a set of atoms such that $Q \subseteq M$, we follow **Branch A**. Then, the system provides a *Proof-based Explanation*, why(q, M, P) answer for every atom $q \in Q$. If the user provides a new query, we go back to the starting point in the flow diagram. If not, we reach an end. For implementing this branch using `xclingo`, the set of atoms Q would be specified by `show_trace` annotations. Then, each why(q, M, P) answer would be implemented following:

$$\text{why}(q, M, q) = \pi_G(q) \text{ such that } G \in G_{P,M}$$

where $G_{P,M}$ is the set of all explanations of M under P (as defined in Definition 6), and $\pi_G(q)$ is the proof for p induced by some G (as defined in Definition 8). Intuitively the default explanations provided by `xclingo` for all the atoms q for model M.

If Q is a formula such that $M \not\models Q$, we follow **Branch B**. From now on, it is supposed that the user expects the existence of a model $M' \neq M$ such that $M' \models Q$. In other words, the user has in mind the existence of an alternative model or solution for which the query holds. The finding of such model M' can be easily implemented as the solving of the program $P \cup \{\bot \leftarrow \neg Q\}$ such that it minimizes the differences between model M' of $P \cup \{\bot \leftarrow \neg Q\}$ and the original model M. Assume for now that such model M' exists, then we continue the flow of **Branch B** and provide a *How To*

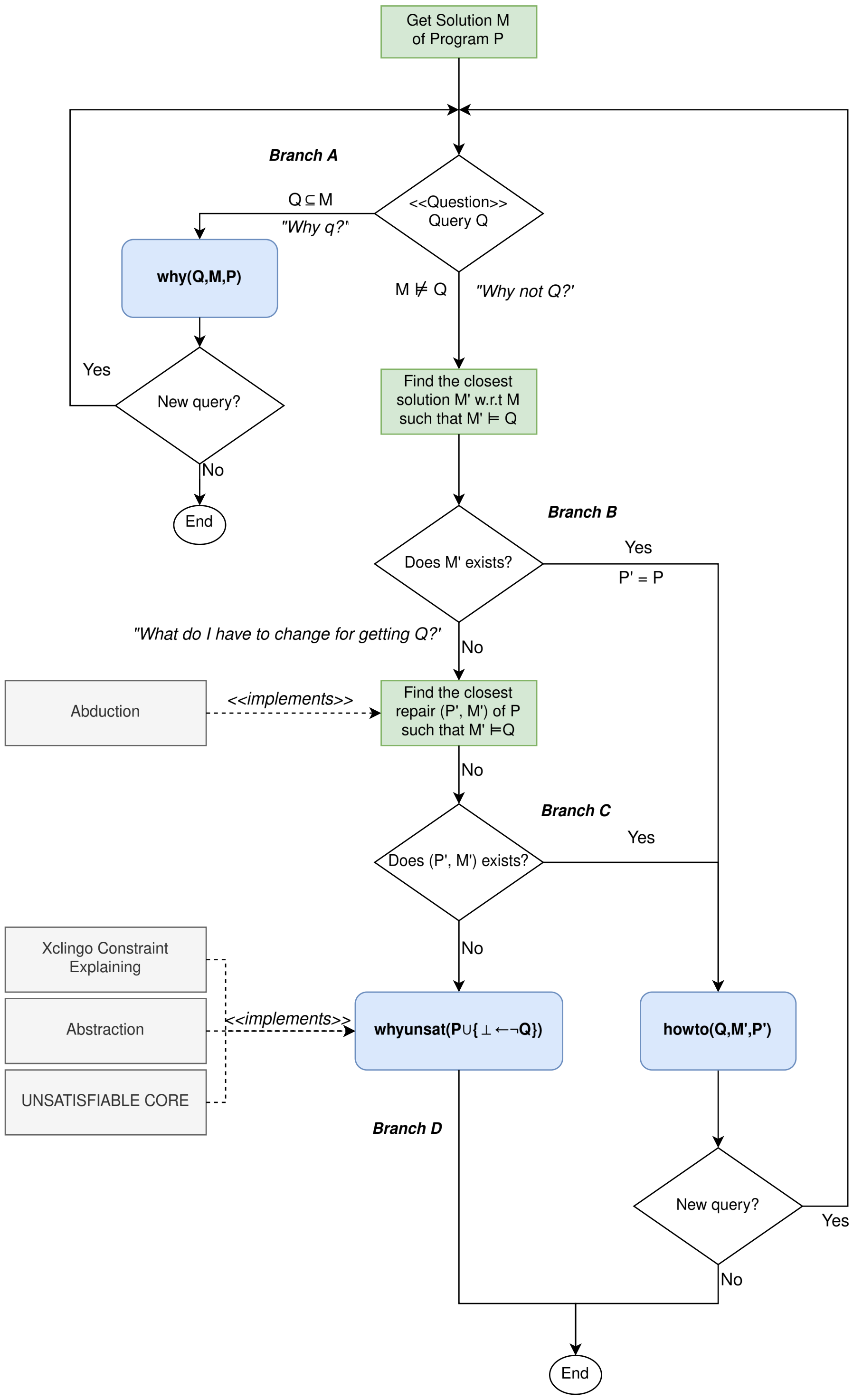


Figure 5.11: Diagram depicting the proposed methodology for answering explanation questions.

Explanation, howto(Q, M, P) as an answer. Implementing this branch using `xclingo`, would not be possible in the sense that Q is a formula and `xclingo` can only receive sets of atoms as a query. In the cases that the formula Q could be mapped to a set of atoms, then we could use the following approach. Once model M' is found, it would be explained using program P via model feeding (see Section 5.4). Formula Q would be translated into a set of atoms, that would be explained as usual with `xclingo`. Thus, the answer howto(Q, M, P) of **Branch B** would be implemented by `xclingo` as:

$$\text{howto}(Q, M', P') = \{\pi_G(q) \mid q \in Q \wedge G \in G_{P',M'}\}$$

where $G_{P,M}$ is the set of all explanations of M under P (as defined in Definition 6), and $\pi_G(q)$ is the proof for p induced by some G (as defined in Definition 8). Intuitively, `xclingo` would compute every explanation graph G for model M' and then would explain each atom $q \in Q$ as a proof-based explanation concerning each graph G.

Branch C on the other hand, corresponds to the case that $SM(P \cup \{\bot \leftarrow \neg Q\}) = \emptyset$. That is, a model M' of P such that $M \models Q$ does not exist. In this case, we look for a repair P', such that there exists a model M' of P' and $M' \models Q$. In particular, we look for the closest possible repair (i.e. such that it requires the fewest changes). This could be implemented by using abductive reasoning. If such repair is possible, we continue the flow of **Branch C** and provide a how-to explanation, howto(Q, M, P) as an answer. If the user provides a new query about M' or M, we go back to the start of the execution. If not, the execution reaches an end. We provide an implementation for **Branch C** using `xclingo` in Section 5.5.3.

If we cannot find a repair, we follow **Branch D**. Where we will provide the user with the reasons for the unsatisfiability of the program P with respect to query Q. To that aim, we need to provide an unsatisfiability explanation, whyunsat$(P \cup \{\bot \leftarrow \neg Q\})$ answer. After that, if the user provides a new query, we go back to the start of the execution. If not, the execution reaches an end.

For showing how **Branch D** can be implemented with `xclingo` please recall Section 4.2.7, where we show how the tool can find reasons for explaining the unsatisfiability of a program, although the definitions given in Chapter 3 do not provide such a notion. As explained in Section 4.3.6, this is done by first disabling the *traced* constraints, by adding new auxiliary head atoms, thus becoming usual rules, and then computing proof-like explanations for the auxiliary atoms derived from such rules. To define this implementation, we could first define the program P' resulting in relaxing some constraints C in P, by adding some arbitrary heads to the rules given by a function γ. More formally, let P be a program such that $SM(P) = \emptyset$ and C be a set of of rules of the form $\bot \leftarrow B$ such that $C \subseteq P$ and $\gamma : C \rightarrow H$ an injective function where H is a set of atoms different from any atom used in P, then $(P', \gamma) = \text{relax}(P, C)$ where P' is the program resulting of replacing any rule $r \in C \subseteq P$ of the form $\bot \leftarrow B$ by a fresh rule r' of the form $\gamma(r) \leftarrow B$.

Thus, given $(P', \gamma) = \text{relax}(P, C)$ where P is an unsatisfiable program and C a set of relaxed constraints in P, and a model M' of P', a whyunsat(P) answer by `xclingo` is a set of proof-based explanation that follows:

$$\text{whyunsat}(P) = \{\pi_G(p) \mid r \in C \wedge G \in G_{P,M}\}$$

where $G_{P,M}$ is the set of all explanations of M under P. Intuitively, the answer is the set of all `xclingo`'s proof-based explanations for the heads introduced for relaxing the constraints. Alternatively, abstraction [90] methods could be used to identify the parts of the program for *catching unsatisfiability reasons of programs* as explained in [89]. The identified abstraction of the program could be provided as an explanation, thus being considered as a technical explanation. Or perhaps that abstraction could be used for generating some kind of proof-like explanation. Finally, another option would be using the notion of (minimal) *Unsatisfiability Core* [103] to find a set of assumptions that lead to an unsatisfiable result. Again, the set of assumptions could be considered the explanation of the unsatisfiability or could be exploited to produce proof-like explanations, by the use of an explanation ASP tool. This approach has been tested using `xclingo` in [83], for obtaining natural language explanations for configuration problems.

5.5.3 Counterfactuals in `xclingo`

A counterfactual *"asserts that if the world had been different in certain specified ways, then things would have been different in other specified ways"*. Following this definition from Judea Pearl [80], we can easily see how **Branch C** is indeed a counterfactual. We face a world where Q does not hold (in any solution), so we imagine a different world where query Q is possible. In particular, we are interested in the closest possible hypothetical world, that is the one that we need the fewest changes to be made to get it. In this section, we show how to perform this counterfactual reasoning, and to obtain explanations for it using `xclingo`. To do so, we refer back to an example that we have mentioned several times in this thesis.

Recall Program 4.2 and consider now its variation in programs 5.6 and 5.7.

```
person(gabriel).
person(clare).

drive(gabriel).
drive(clare).

alcohol(gabriel, 40).
alcohol(clare, 5).

resist(gabriel).
```

Program 5.6: Data for gabriel and clare.

```
%!trace_rule {"% drove drunk (over 30mg/l)", P}.
punish(P) :- drive(P), alcohol(P,A), A>30, person(P).

%!trace_rule {"% resisted to authority", P}.
punish(P) :- resist(P), person(P).

%!trace_rule {"% is innocent by default",P}.
sentence(P, innocent) :- person(P), not punish(P).

%!trace_rule {"% has been sentenced to prison", P}.
sentence(P, prison) :- punish(P).
```

Program 5.7: Rules for being punished and sentenced.

The laws for punishing and sentencing persons in Program 5.7 are the with respect to Program 4.2. On the contrary, the data talking about `clare` and `gabriel` does change in Program 5.6 in the sense that we removed the choice deciding wether `gabriel` resists authority or drinks too much alcohol. Now, both things are true, so the program only has one answer set, where `clare` is innocent, and `gabriel` has two alternative reasons for going to prison shown in Output 5.6.

Imagine now a user receives this output from a system implementing the methodology presented in Section 5.5. This user expected `clare` to be sentenced to prison, so immediately asks for another *possible* world where query Q, `sentence(clare, prison)` is true. This is **Branch B**. However, this is not possible given the current events (i.e. given the facts provided in Program 5.6 there is only one answer set). The system informs of this and starts to imagine different scenarios (i.e. changing the actual facts),

```
##Explanation: 1.1
  *
  |__"gabriel has been sentenced to prison"
  |  |__"gabriel drove drunk (over 30mg/l)"

  *
  |__"clare is innocent by default"
```

```
##Explanation: 1.2
  *
  |__"gabriel has been sentenced to prison"
  |  |__"gabriel resisted to authority"

  *
  |__"clare is innocent by default"
```

Output 5.6: Output by `xclingo` for programs 5.6 and 5.7 when we ask to explain sentences for clare and gabriel.

trying to find the least set of changes that need to be made so `clare` enters prison. Therefore, entering in **Branch C**.

This implies performing abductive reasoning about what `gabriel` and `clare` did (or did not), so that the user's expected situation would have happened. For instance, by considering a scenario where `gabriel` had not resisted authority, forgetting events that occurred, or envisioning `clare` consuming enough alcohol to be punished, reflecting on hypothetical situations. This abductive reasoning is performed by Programs 5.8.

```
1  %%% addable
2  _abducible(alcohol(clare, 35)).        _abducible(resist(clare)).
3
4  %%% removable
5  _abducible(drive(gabriel)).            _abducible(drive(clare)).
6  _abducible(alcohol(clare, 5)).         _abducible(alcohol(gabriel, 40)).
7  _abducible(resist(gabriel)).
8
9  {_abduced(add,A): not model(A)} :- _abducible(A).
10 {_abduced(rm,A): model(A)} :- _abducible(A).
11 #minimize{1,A: _abduced(T,A)}.
```

Program 5.8: Performing abduction over the events for imagining different worlds.

It is assumed that the system is somehow provided with a set, indicating which atoms are *abducbile*. This could be fixed or dynamically specified by the user depending on the application. In the case of `xclingo`, this could be easily implemented in the form of a, say, `%!abducible` annotation, that would work as the other two annotations that define sets over atoms (i.e. `trace` and `mute`). For instance here, driving, drinking and resisting are abducible, but `person` is not included as an abducible predicate. The choice rules in lines 9 and 10 generate the different abduced atoms. In particular, the first one abduces atoms that are not in the reference model (i.e. M in Figure 5.11 from Section 5.5), this is `_abduced(add, A)`. Whereas the second one abduces atoms that are true in the model, respectively `_abduced(add, A)`. Note that `model(A)` represents the `_xclingo_model` predicate explained in section 4.3.1. Finally, the `#minimize` in the last line, ensures we find the closest hypothetical world possible.

Now, to connect the rules back to the punish and sentence rules in Program 5.7, we use the rules in Program 5.9. Intuitively, we take atoms that were in the model and not removed and those that were not in the model and abduced. The programs from figures 5.7, 5.9 and 5.9 form our program P' in our methodology (Figure 5.11).

```
model(person(gabriel)).      model(person(clare)).
model(drive(gabriel)).       model(drive(clare)).
model(alcohol(gabriel, 40)). model(alcohol(clare, 5)).
model(resist(gabriel)).

person(P) :- model(person(P)), not _abduced(rm, person(P)).
person(P) :- _abduced(add, person(P)).

drive(P) :- model(drive(P)), not _abduced(rm, drive(P)).
drive(P) :- _abduced(add, drive(P)).

alcohol(P, A) :- model(alcohol(P, A)), not _abduced(rm, alcohol(P, A)).
alcohol(P, A) :- _abduced(add, alcohol(P, A)).

resist(P) :- model(resist(P)), not _abduced(rm, resist(P)).
resist(P) :- _abduced(add, resist(P)).
```

Program 5.9: Rules connecting the abduced facts back to the usual predicates.

Finally, we need to put together our query Q and our new program P'. Recalling **Branch C** of Figure 5.11, we are only interested in hypothetical worlds (M') such that the query is true (i.e. $M' \models Q$). In other words, the program that we want to solve is $P' \cup \{\bot \leftarrow \neg Q\}$. Thus, we include a constraint forcing the query in Program 5.10, as well as some annotations for xclingo.

```
:- not sentence(clare, prison).
%!show_trace {sentence(clare, S)}.
%!trace {_abduced(add,A), "_abduction_ Had % held", A}.
%!trace {_abduced(rm,A),  "_abduction_ Hadn't % held", A}.
```

Program 5.10: Introducing query Q

By putting the four pieces of code together and calling xclingo, we obtain the output in Output 5.7.

```
Answer: 2

##Explanation: 2.1
  *
  |__"clare has been sentenced to prison"
  |  |__"clare drove drunk (over 30mg/l)"
  |  |  |__"_abduction_ Had alcohol(clare,35) held"

##Total Explanations:   1
```

```
Answer: 3

##Explanation: 3.1
  *
  |__"clare has been sentenced to prison"
  |  |__"clare resisted to authority"
  |  |  |__"_abduction_ Had resist(clare) held"


##Total Explanations:   1
```

Output 5.7: Output from xclingo for programs 5.7, 5.9, 5.9 and 5.10 together, implementing **Branch C** from Figure 5.11.

This implementation finds two alternative hypothetical worlds where clare goes to prison, namely one in which she resists authority and one where she drinks too much alcohol. The worlds found only add those both atoms (i.e. alcohol(clare, 35) resist(clare)) due to the minimize. As a final remark, the option `-opt-mode=optN` has to be introduced when using `xclingo` for obtaining both minimal hypothetical models.

Chapter 6

Related Work

The topic of explainability in ASP has been explored with a growing interest over the last decade. Only 5 years after the publication of the survey [48], new systems have arisen and some existing approaches have evolved.

In this chapter, we review some of the most relevant approaches of the literature on ASP explanations, including formalisms and tools. We make a comparison between each approach and ours under triple lens. First, we compare the mathematical objects that support the different definitions of explanation. Second, for those approaches that provide usable tools, we compare their functionalities against each other and against `xclingo`. Third, wherever possible, we compare the approaches in terms of the possibility of obtaining commonsense explanations.

6.1 Causal Graphs

Causal Graphs (c-graphs) [19, 49] introduce several concepts that have served as the basis for most of the work explained in this dissertation. More precisely, the concept *causal graphs* are graph-like explanations that serve as the basis behind the idea of support graphs. Indeed `xclingo` started as a partial implementation of causal graphs to obtain causal explanations from labeled logic programs. Causal graphs act as the semantics for *Causal Terms* and *Causal Values* that provide algebraic expressions to represent and evaluate causes (i.e. the graphs) whose properties create a distributive complete lattice. They provide a causal semantics that extends *stable models*, *answer sets* and *well-founded models*, and define *Causal Literals* showing how they can be applied to derive causal information for standard logic programs.

Both definitions, causal graphs and support graphs, start from the concept of *Labeled Logic Programs*, introduced in [19], where each rule in the program is labeled and causes are represented as graphs so that the vertices are those rule labels. Let us start by illustrating causal values's algebraic expressions to represent these graphs. To this aim, product (*) and application (·) are used to represent the edges of a graph. In particular, (*) captures the idea that several causes need to work together to cause an effect, while (·) captures the idea of causal chain. For instance, the following graph (first introduced in Figure 3.1) can be represented by the expression below:

$$\begin{tikzcd}
\ell_1 : p \arrow[r] \arrow[rr, bend left] & \ell_2 : q \arrow[r] & \ell_3 : r
\end{tikzcd}$$

$$\ell_1 \cdot \ell_3 * \ell_1 \cdot \ell_2 \cdot \ell_3$$

Furthermore, the sum (+) operator can be used for separating alternative causes. Thus, the following graphs (initially introduced in Figure 4.1), can be represented by the expression below:

$$\begin{tikzcd}
\ell_1 : resist(gabriel) \arrow[d] & \ell_2 : alcohol(gabriel, 40) \arrow[dr] & \ell_3 : drive(gabriel) \arrow[d] \\
\ell_4 : punish(gabriel) & & \ell_5 : punish(gabriel)
\end{tikzcd}$$

$$\ell_1 \cdot \ell_4 + (\ell_2 * \ell_3) \cdot \ell_5$$

Note that, while in support graphs each graph explains the whole model and we have a different graph for each alternative explanation, here a unique causal value expression captures all alternative explanations for ℓ_5 regarding the same answer set. This is the first difference between both approaches.

Moreover, these expressions fulfill several properties so that they form a distributive complete lattice. In particular, these properties are detailed in Figures 13 to 16 from [49]. Following these properties, we can perform interesting equivalences between causal values. For instance, the following holds:

$$\ell_1 \cdot \ell_4 + (\ell_2 * \ell_3) \cdot \ell_5 = \ell_1 \cdot \ell_4 + (\ell_2 \cdot \ell_5) * (\ell_3 \cdot \ell_5)$$

Furthermore, regarding the first example, we can establish the following equivalence,

$$\ell_1 \cdot \ell_3 * \ell_1 \cdot \ell_2 \cdot \ell_3 = \ell_1 \cdot \ell_2 \cdot \ell_3$$

where one of the joint causes is *subsummed* by the other (stronger) cause. This constitutes a second difference with respect to support graphs, where having this property would mean removing the edge (ℓ_1, ℓ_3) in the graph.

Let us now introduce the different notions of graphs proposed by Fandinno et al. and how they relate to causal values and support graphs. Consider for instance Program 3.3 from [49] as an example (see Program 6.1).

$$\begin{array}{rlrl}
b : & bomb \leftarrow open & s : & wireless \\
u : & open \leftarrow up(a), up(b) & y : & wireless \\
l(L) : & up(L) \leftarrow wireless & &
\end{array}$$

Program 6.1: Program 3.1 from [49].

In the example, a suitcase containing a bomb can be activated by a wireless mechanism that flips two locks ($up(a)$, and $up(b)$), causing the bomb to explode. The mechanism is activated simultaneously by two different remote controls s and y. The causal value cv_{bomb} for b is:

$$cv_{bomb} = ((s + y) \cdot l(b) * (s + y) \cdot l(a)) \cdot u \cdot b$$

Which can be rewritten in Disjunctive Normal Form (DNF), that is, as a sum of terms using only (*) and (·):

$$\begin{aligned} cv_{bomb} = \quad & s \cdot (l(a) * l(b)) \cdot u \cdot b + \\ & y \cdot (l(a) * l(b)) \cdot u \cdot b + \\ & (s \cdot l(a) * y \cdot l(b)) \cdot u \cdot b + \\ & (y \cdot l(a) * s \cdot l(b)) \cdot u \cdot b \end{aligned}$$

Each term in the expression of a causal value in DNF form is defined by Fandinno et al. as a *Sufficient Cause*.

Recalling Definition 3.2 from [49], causal graphs are sets of edges $G \subseteq Lb \times Lb$ transitively and reflexively closed, where Lb is the set of labels from a logic program. For instance, Figure 6.1 depicts the c-graphs corresponding to the four possible proofs for atom *bomb* in Program 6.1.

$$\begin{tikzcd}
& s \arrow[dl] \arrow[dr] & & & y \arrow[dl] \arrow[dr] & & s \arrow[d] & & y \arrow[d] & y \arrow[d] & & s \arrow[d] \\
l(a) \arrow[dr] & & l(b) \arrow[dl] & l(a) \arrow[dr] & & l(b) \arrow[dl] & l(a) \arrow[dr] & & l(b) \arrow[dl] & l(a) \arrow[dr] & & l(b) \arrow[dl] \\
& u \arrow[d] & & & u \arrow[d] & & & u \arrow[d] & & & u \arrow[d] & \\
& b & & & b & & & b & & & b & \\
& CG_1 & & & CG_2 & & & CG_3 & & & CG_4 &
\end{tikzcd}$$

Figure 6.1: C-graphs corresponding to the four different proofs of atom *bomb* in (labeled) Program 6.1.

Whose corresponding causal values would be:

$$\begin{aligned} cv_1 &= s \cdot (l(a) * l(b)) \cdot u \cdot b & \quad cv_3 &= (s \cdot l(a) * y \cdot l(b)) \cdot u \cdot b \\ cv_2 &= y \cdot (l(a) * l(b)) \cdot u \cdot b & \quad cv_4 &= (y \cdot l(a) * s \cdot l(b)) \cdot u \cdot b \end{aligned}$$

And the sum of all of them equivalates to cv_{bomb},

$$cv_1 + cv_2 + cv_3 + cv_4 = cv_{bomb}$$

meaning that they are the set of sufficient causes for b. Moreover, although the c-graphs $(CG_1, \ldots, CG_4)$ in the figure correspond to the transitive and reflexive reductions of the real c-graphs, both the transitivity and reflexivity properties are of great importance as they allow the graphs to be expressed as just a set of edges, and thus establish an order $\subseteq$ relation between them. Indeed, the four c-graphs from the figure are $\subseteq$-minimal, only differing in the presence of edges $(s, l(a))$, $(s, l(b))$, $(y, l(a))$ and $(y, l(b))$. However, although the $\subseteq$-minimality between c-graphs is not enough to capture this, it is clear that graphs CG_3 and CG_4 are redundant with respect to CG_1 and CG_2.

To deal with this redundancy, *Proof Graphs* $cgraph(\pi)$ are introduced as c-graphs that correspond to a particular proof π of an atom obtained by first labeling each atom in the program and then forming edges connecting: (1) subproof consequences with proof (i.e. rule) labels; and (2) these labels in turn with its corresponding consequent's atom label. For instance, consider the *Proof Graphs* for atom *bomb* depicted in Figure 6.2.

$$\begin{tikzcd}
 & s \arrow[d] & \\
 & wireless \arrow[dl] \arrow[dr] & \\
l(a) \arrow[d] & & l(b) \arrow[d] \\
up(a) \arrow[dr] & & up(b) \arrow[dl] \\
 & open \arrow[d] & \\
 & u \arrow[d] & \\
 & b \arrow[d] & \\
 & bomb &
\end{tikzcd}$$

PG_1

$$\begin{tikzcd}
 & y \arrow[d] & \\
 & wireless \arrow[dl] \arrow[dr] & \\
l(a) \arrow[d] & & l(b) \arrow[d] \\
up(a) \arrow[dr] & & up(b) \arrow[dl] \\
 & open \arrow[d] & \\
 & u \arrow[d] & \\
 & b \arrow[d] & \\
 & bomb \;\; P &
\end{tikzcd}$$

PG_2

$$\begin{tikzcd}
s \arrow[dr] & & y \arrow[dl] \\
 & wireless \arrow[dl] \arrow[dr] & \\
l(a) \arrow[d] & & l(b) \arrow[d] \\
up(a) \arrow[dr] & & up(b) \arrow[dl] \\
 & open \arrow[d] & \\
 & u \arrow[d] & \\
 & b \arrow[d] & \\
 & bomb &
\end{tikzcd}$$

$PG_{3,4}$

Figure 6.2: *Proof Graphs* corresponding to the four different proofs of atom *bomb* in (labeled) Program 6.1.

$$\begin{tikzcd}
 & s : wireless \arrow[dl] \arrow[dr] & \\
l(a) : up(a) \arrow[dr] & & l(b) : up(b) \arrow[dl] \\
 & u : open \arrow[d] & \\
 & b : bomb &
\end{tikzcd}$$

SG_1

$$\begin{tikzcd}
 & y : wireless \arrow[dl] \arrow[dr] & \\
l(a) : up(a) \arrow[dr] & & l(b) : up(b) \arrow[dl] \\
 & u : open \arrow[d] & \\
 & b : bomb &
\end{tikzcd}$$

SG_3

Figure 6.3: *Support Graphs* for Program 6.1.

Note how PG_1 and PG_2 respectively correspond with CG_1 and CG_2, whereas $PG_{3,4}$ corresponds to both CG_3 and CG_4. A *Redundant Proof* is then defined as a proof so that there exists another proof whose subproofs are a strict subset of the latter. This notion can be easily captured in proof graphs by $\subseteq$-minimality among their sets of edges. For instance $PG_{3,4}$ is redundant (is a strict superset of) with respect to PG_1 and PG_2. Finally, by only taking the non-redundant proof graphs (PG_1 and PG_2) and ignoring any atom vertex, we end up having CG_1 and CG_2 again. These two graphs correspond to the support graphs we would obtain by applying the Definition 6 from Section 3.2 to Program 6.1. They are shown in Figure 6.3.

In particular, SG_1 and SG_2 correspond to PG_1 and PG_2 respectively, and CG_1 and CG_2 respectively. In contrast, $PG_{3,4}$, CG_3 and CG_4 have no correspondant support graphs. This naturally follows from the support graphs definition. In those three graphs two different rules (those labeled with s and y) are used to give support to the same atom $wireless$ (see $PG_{3,4}$), which is not allowed in support graphs by condition (i) of Definition 6. This establishes a (third) important difference between support graphs and c-graphs, that is, the support graphs represent a subset of the Fandinno et al's sufficient causes.

Also regarding the definition of support graphs, note that c-graphs are only defined for non-disjunctive programs whereas our approach handles disjunctive programs. In contrast, support graphs only consider programs with a unique labeling for the rules, that is, the labels act as rule identifiers. Meanwhile, in c-graphs one can repeat labels among rules, meaning that the same label can justify different atoms. Finally, and also related to that, in c-graphs we can also filter irrelevant information out from the explanations. This is done by labeling the rules with a 1, which is the neutral element in the causal values. After simplifying the expressions, the labels disappear, but the transitive causal connections are preserved, just as it happens in support graphs with the forgetting operation (see Definition 10).

6.2 And-Or Explanation Trees and Tree Explanations

In [45], Erdem and Öztok developed a system for query answering and explanation of biomedical queries. The queries are stated in a controlled natural language called *BioQuery-CNL** [46, 78] and translated into ASP programs. Biomedical knowledge is extracted from RDF ontologies and then translated into an ASP program [13, 40]. The query is answered by solving the ASP program resulting from the union of both the query encoding and the logic program. Besides, this system is incremented with the embedding of a tool called ExpGen-ASP [43, 78] for generating natural language explanations for the answers to the queries. This section studies the explanations generated by that system and its relation to support graphs and causal graphs.

Extracting Shortest Explanations from And-Or Explanation Trees

In [45], the notion of explanation is defined as a *Vertex Label Tree* with respect to an ASP program and a particular answer set of that program. Vertices from those trees are labeled either with the rules (namely *rule-vertices*) from the program or the atoms (namely *atom vertices*) in the answer set. Resembling causal proof graphs from [49] (see Section 6.1), they define a special case of vertex label trees called *and-or explanation trees* (Definition 2 from [45]). In them, edges go from rule vertices to the atom vertices whose corresponding atoms are supported by that particular rule, and then from the atoms in the positive body of the rules to their corresponding rule vertices. Furthermore, they are defined starting from a given atom p (belonging to the explained answer set), which labels the vertex in the root of the tree. For instance, Figure 6.4 shows the and-or explanation tree obtained for atom *bomb* in Program 6.1.

Note that each one of the two rules "$wireless \leftarrow$" supporting $wireless$ atom vertex in both branches of $AOET_1$ come from different rules from Program 6.1, respectively labeled with s and y. Since their representation does not start from labeled logic programs imagine, for the sake of the comparison, that Program 6.1 is modified according to the following rules:

$$\begin{array}{ll} wireless \leftarrow s\_wireless & s\_wireless \leftarrow \\ wireless \leftarrow y\_wireless & y\_wireless \leftarrow \end{array}$$

We draw these two labels here for clarity, although this representation of explanations does not start from labeled logic programs. Intuitively, *rule vertices are AND vertices* whereas *atom vertices are OR*

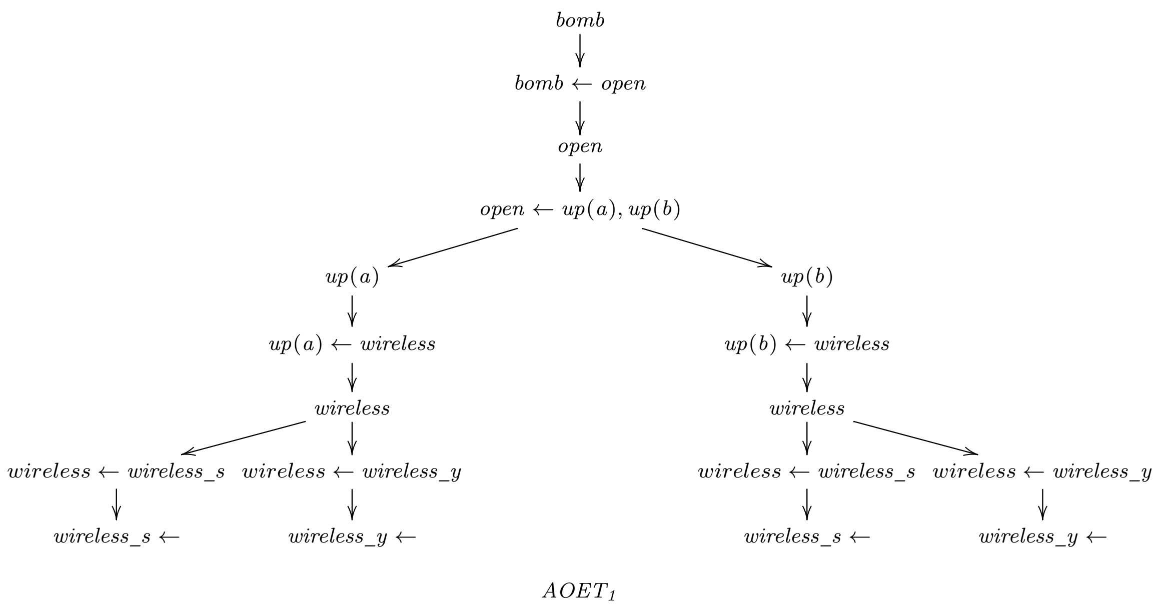


Figure 6.4: And-or explanation tree for atom *bomb* corresponding to Program 6.1 and its unique answer set.

vertices meaning that each supporting rule constitutes an alternative cause. In a sense, the and-or trees contain every possible explanation for an atom with respect to a particular answer set. Indeed, they define the notion of *Explantion Tree* which are extracted from and-or explanation trees by (informally speaking) visiting only one child for every OR vertex to produce several *non-disjunctive trees* each one representing a particular explanation for the atom. This notion coincides with the relation between causal values and sufficient causes from [49]. To illustrate this, Figure 6.5 shows the 4 explanation trees that are obtained from $AOET_1$.

Note how each explanation tree corresponds to one causal proof graph from Figure 6.2, but replacing the label of each rule vertex by the corresponding rule label in Program 6.1. In particular ET_1 corresponds to PG_1, ET_3 corresponds to PG_2, and ET_2 and ET_4 corrspond to PG_4. Additionally, they include the notion of *Explanation* which results in *ignoring* rule vertex in each explanation tree, removing them from the trees and leaving only the atom vertices. Again, this is a deletion in the same sense that applying the *forgetting* operation from Definition 10. Finally, they introduce an ordering among explanations based on the cardinality of the set V of vertex of each tree, defining as *shortest explanations* those with the minimal number of vertex. The system ExpGen-ASP uses two different algorithms (provided in [45]) to either generate one or (respectively) k shortest explanations.

Interestingly, the set of explanation trees obtained for a particular atom and answer set seems to have a one-to-one correspondence with the sufficient causes from [49]. In particular, explanations seem to be a special case of proof graphs where the whole rule is introduced as a node in the graph, instead of using a label that identifies the rule.

$$\begin{tikzcd}
 & \mathit{bomb} \arrow[d] & \\
 & \mathit{bomb} \leftarrow \mathit{open} \arrow[d] & \\
 & \mathit{open} \arrow[d] & \\
 & \mathit{open} \leftarrow \mathit{up}(a), \mathit{up}(b) \arrow[dl] \arrow[dr] & \\
\mathit{up}(a) \arrow[d] & & \mathit{up}(b) \arrow[d] \\
\mathit{up}(a) \leftarrow \mathit{wireless} \arrow[d] & & \mathit{up}(b) \leftarrow \mathit{wireless} \arrow[d] \\
\mathit{wireless} \arrow[d] & & \mathit{wireless} \arrow[d] \\
\mathit{wireless} \leftarrow \mathit{wireless\_s} \arrow[d] & & \mathit{wireless} \leftarrow \mathit{wireless\_s} \arrow[d] \\
\mathit{wireless\_s} \leftarrow & & \mathit{wireless\_s} \leftarrow
\end{tikzcd}$$

ET_1

$$\begin{tikzcd}
 & \mathit{bomb} \arrow[d] & \\
 & \mathit{bomb} \leftarrow \mathit{open} \arrow[d] & \\
 & \mathit{open} \arrow[d] & \\
 & \mathit{open} \leftarrow \mathit{up}(a), \mathit{up}(b) \arrow[dl] \arrow[dr] & \\
\mathit{up}(a) \arrow[d] & & \mathit{up}(b) \arrow[d] \\
\mathit{up}(a) \leftarrow \mathit{wireless} \arrow[d] & & \mathit{up}(b) \leftarrow \mathit{wireless} \arrow[d] \\
\mathit{wireless} \arrow[d] & & \mathit{wireless} \arrow[d] \\
\mathit{wireless} \leftarrow \mathit{wireless\_s} \arrow[d] & & \mathit{wireless} \leftarrow \mathit{wireless\_y} \arrow[d] \\
\mathit{wireless\_s} \leftarrow & & \mathit{wireless\_y} \leftarrow
\end{tikzcd}$$

ET_2

$$\begin{tikzcd}
 & \mathit{bomb} \arrow[d] & \\
 & \mathit{bomb} \leftarrow \mathit{open} \arrow[d] & \\
 & \mathit{open} \arrow[d] & \\
 & \mathit{open} \leftarrow \mathit{up}(a), \mathit{up}(b) \arrow[dl] \arrow[dr] & \\
\mathit{up}(a) \arrow[d] & & \mathit{up}(b) \arrow[d] \\
\mathit{up}(a) \leftarrow \mathit{wireless} \arrow[d] & & \mathit{up}(b) \leftarrow \mathit{wireless} \arrow[d] \\
\mathit{wireless} \arrow[d] & & \mathit{wireless} \arrow[d] \\
\mathit{wireless} \leftarrow \mathit{wireless\_s} \arrow[d] & & \mathit{wireless} \leftarrow \mathit{wireless\_s} \arrow[d] \\
\mathit{wireless\_y} \leftarrow & & \mathit{wireless\_s} \leftarrow
\end{tikzcd}$$

ET_3

$$\begin{tikzcd}
 & \mathit{bomb} \arrow[d] & \\
 & \mathit{bomb} \leftarrow \mathit{open} \arrow[d] & \\
 & \mathit{open} \arrow[d] & \\
 & \mathit{open} \leftarrow \mathit{up}(a), \mathit{up}(b) \arrow[dl] \arrow[dr] & \\
\mathit{up}(a) \arrow[d] & & \mathit{up}(b) \arrow[d] \\
\mathit{up}(a) \leftarrow \mathit{wireless} \arrow[d] & & \mathit{up}(b) \leftarrow \mathit{wireless} \arrow[d] \\
\mathit{wireless} \arrow[d] & & \mathit{wireless} \arrow[d] \\
\mathit{wireless} \leftarrow \mathit{wireless\_y} \arrow[d] & & \mathit{wireless} \leftarrow \mathit{wireless\_y} \arrow[d] \\
\mathit{wireless\_y} \leftarrow & & \mathit{wireless\_y} \leftarrow
\end{tikzcd}$$

ET_4

Figure 6.5: The four *Explanation Trees* obtained from $AOET_1$ depicted in Figure 6.4.

Relevant Part of a Program Concerning a Query

One important trait of the approach followed by Erdem and Öztok [45] is that, for obtaining explanations for an atom p, they start from the *relevant part of the program* with respect to p. In other words, they discard those rules in the program that are not useful for answering queries about p, thus linearly reducing the computation time for processing such queries. The particular method to identify the relevant rules is given in [43]. The intuition is to find the reachable symbols in the *Predicate Dependency Graph* from [12] of the given program π (namely $DG(\pi)$) starting from atom p. Any rule in the original program π whose head contains a reachable atom is part of the relevant set of rules, where *reachability* is considered as the transitive closure over $DG(\pi)$.

Although the notion of support explanation graphs given in Chapter 3 does not include any notion of *relevance* of a labeled logic program, the `xclingo` uses a similar technique, as explained at the beginning of Section 4.3.4.

Explanation Design with ExpGen-ASP

As for `xclingo` and most of the explanation tools for ASP, the structure of the explanations obtained from ExpGen-ASP fundamentally emerges from (and its binded to) the rules of the particular ASP program. Although, Erdem and Öztok's definition of explanation is neither explicitly built from nor related to any notion of *derivation* or *proof*, their tree structure resembles one. Indeed, we have already seen how a one to one correspondence can be established to the proofs extracted from a support graph, and thus to Modus Ponens derivations following Proposition 2.

In [45] and [78], the authors mention the possibility of expressing the explanations in natural language. This feature has been added to the ExpGen-ASP tool as a way to facilitate the understanding of explanations obtained for particular queries, providing. Figure 6.6 shows an example of a natural language explanation of a shortest explanation for a particular query.

```
The distance of the gene CASK from the start gene is 2.
    The gene CASK interacts with the gene DLG4 according to BioGRID.
    The distance of the gene DLG4 from the start gene is 1.
        The gene DLG4 interacts with the gene ADRB1 according to BioGRID.
        ADRB1 is the start gene.
```

Figure 6.6: An example of a natural language explanation (from [45]) produced by the BioQuery-ASP system after the embedding of the ExpGen-ASP system. The explanation is meant to explain the query "*What are the genes that are related to the gene ADRB1 via gene-gene interaction chain of length at most 3?*". The answer to the query is "CASK".

Resembling how `xclingo` displays the information, the text is structured in different levels of indentation where each new indentation can be read as a "because". To generate such text, the tool allows the user to define a set of template-like natural language expressions that are associated with particular predicates. According to [45], this table is not meant to define a text for every predicate used in the explanations and thus, only some of the vertices have an associated text. They also mention that a preorder depth-first traversal of the trees collects such vertices and, from that, the text is then built using the look-up table, although this process is not detailed. From the results shown, it could be assumed that vertices without text are *ignored* or *forgotten* in a similar way that `xclingo` deals with non-annotated atoms (that is, following the *forgetting* operation defined in Definition 10).

This feature provides the KR-engineer with some flexibility to design explanations. However, it seems the text templates can only be defined at a predicate level, that is, all the atoms from a particular predicate are associated with the defined text (see Table 4 from [45]). Of course, this proves to be sufficient to deal with the particular case of answering biomedical queries. However, it may lack the expressiveness needed to conveniently design accountable explanations for other real case scenarios such as problem-solving, planning or diagnosis where, under particular KR representations, atoms belonging to the same predicate could need to be ignored or not, or may need to be associated with different texts. In Chapter 5, we have shown the impact of the use of such explanation design features when trying to fulfill end-user explanation needs.

6.3 s(CASP)

Minimal s(CASP) Justification Trees

The system s(ASP), presented in [70], introduces a method for computing stable models of a normal logic program without requiring it to be grounded. Therefore, several additional properties emerge such as the use of variables that may range over infinite domains or the use of complex data structures like lists. s(CASP) [9] represents an incremental enhancement of s(ASP), extending it with Constraint ASP programs. The evaluation method is a query-driven (goal-directed) execution and backward chaining that computes *partial answer sets*. For instance, consider Program 6.2 taken from the s(CASP) tutorial in [10].

```
1  opera(saturday) :- not home(saturday).
2  home(saturday)  :- not opera(saturday).
3  dinner(sunday).
4
5  ?- opera(D).
```

Program 6.2: Example 4 from [10].

This program has two answer sets, namely $A_1 = \{dinner(sunday), opera(saturday)\}$ and $A_2 = \{dinner(sunday), home(saturday)\}$. The clause starting with `?-` in line 5 represents the query that s(CASP) has to find an answer for, although the tool can be queried on the fly if called in *interactive* mode. For the query `opera(D)` and Program 6.2, s(CASP) returns the one partial answer set $\{nothome(saturday), opera(saturday)\}$. A partial answer set can be defined as a subset of an answer set of the program that contains only what is needed to answer a given query. Note that a partial answer set potentially represents or relates to several (always at least one) answer sets. For this particular case, it relates only to the original answer set A_1 for which it is true that $opera(saturday) \in A_i and home(saturday) \notin A_i$. Also note that $dinner(sunday)$ is not present in the partial answer set, although it belongs both to A_1 and A_2, because it is not relevant for answering query `opera(D)`.

Another important difference with respect to other ASP systems that also emerges from this top-down evaluation method is that *safety* does not need to be ensured anymore, that is, variables used in the scope of default negated atoms `not p(X)` do not need to appear in any positive literal. For instance, consider Program 6.3 taken from Example 7 from [10].

```
opera(X) :- not home(X).
home(X)  :- not opera(X).
home(monday).

:- baby(D), opera(D).

baby(tuesday).

?- opera(D).
```

Program 6.3: Example 7 (and Figure 1) from [10].

This program cannot be grounded by a solver like `clingo`, due to the *unsafety* of variable `X` in lines 1 and 2. However, for query `opera(D)`, s(CASP) produces the following partial answer set.

$$\begin{aligned}\{&\text{baby(tuesday), home(tuesday), not opera(tuesday)},\\ &\text{not baby}(\_|\{\_ \notin [\text{tuesday}]\}),\\ &\text{not home}(D|\{D \notin [\text{monday,tuesday}]\}),\\ &\text{opera}(D|\{D \notin [\text{monday,tuesday}]\})\}\end{aligned}$$

Note how, for instance, the literal not home$(D \mid \{D \notin [\text{monday,tuesday}]\})$ is defined in terms of the set of values variable D cannot be bound to. This may happen as well with positive literals like opera$(D \mid \{D \notin [\text{monday,tuesday}]\})$.

Additionally, s(CASP) counts with the capacity to produce natural language explanations for the results of the queries [8]. Authors call these explanations *Minimal s(CASP) Justification Trees*. These trees are constructed by recording the literals in the path of a successful goal-driven proof of a given query. They are obtained together with the answer of a query when the option `-tree` is enabled. Unlike support graphs and the rest of previously commented approaches, that obtain explanations using an answer set as a starting point, s(CASP) justifications are so with respect to a particular partial answer set. For Program 4.1, using `xclingo`'s option `-auto-tracing=all` and the following `show_trace` annotation,

```
%!show_trace sentence(P,S) :- sentence(P,S).
```

we obtained the explanations in Output 4.7. Since s(CASP) does not share the syntax for `clingo` *choice rules*, the loop in Figure 6.7 can be introduced instead to create the three scenarios of Program 4.1, in order to compare the explanations of both systems.

This introduces new predicates `w1`, `w2` and `w3` that, of course, will affect the explanations. For instance, Output 6.1 shows the justification that s(CASP) provided accompanying the first answer for the query `?- sentence(P,S)`.

The output is divided into three parts starting by showing the justification tree, then providing the partial answer set and finally the bindings for the query's free variables. The justification is shown in a similar way as `xclingo` proof for an atom, where the subsequent proofs (or reasons) for each atom are implicitly indicated by the level of indentation in the text, and in the s(CASP) case explicitly shown by the $\leftarrow$ and the $\wedge$ symbols. The second part of the explanation relates to the justification of the

```
%1{alcohol(gabriel, 40); resist(gabriel)}.
w1 :- not w2, not w3.
w2 :- not w1, not w3.
w3 :- not w1, not w2.
alcohol(gabriel, 40) :- w1.
resist(gabriel) :- w2.
alcohol(gabriel, 40) :- w3.
resist(gabriel) :- w3.
```

Figure 6.7: Loop introduced in Program 4.1 to replicate the effect of the choice rule commented in line 1 (originally in line 6).

```
% Justification
query ←
  sentence(gabriel,prison) ←
      punish(gabriel) ←
          drive(gabriel) ∧
          alcohol(gabriel,40) ←
          w1 ←
              not w2 ←
                  chs(w1) ∧
              not w3 ←
                  chs(w1) ∧
          40>30 ∧
          person(gabriel) ∧
  o_nmr_check ←
      not o_chk_1 ←
          proved(not w2) ∧
          proved(not w3) ∧
          proved(w1) ∧
      not o_chk_2 ←
          proved(w1) ∧
      not o_chk_3 ←
          proved(w1).
% Model
{ punish(gabriel), sentence(gabriel,prison)}
% Bindings
P = gabriel,
S = prison
```

Output 6.1: One of the justifications provided by s(CASP) to for Program 4.1 after replacing the choice rule by the loop in Figure 6.7

global constraints predicate (namely `o_nmr_check`). Those are constraints introduced by the s(CASP) compiler to ensure the partial models are consistent with the user's explicitly written constraints but also with loops (see *Loop Handling* in [7]) as happens in the code shown in Figure 6.7. For the sake of the comparison, the parts of the justifications concerning the introduced loop (namely the global constraint justification and any allusions to literals `w1`, `w2` and `w3`) will be omitted from now on.

To obtain all the justifications we can use the Command 6.1,

```
scasp -tree -s0 -query="sentence(P,S)" dont_drive_drunk.lp
```
(Command 6.1)

where the option `-s0` tells the system to obtain all the possible partial answer sets for the query, similarly to `-n` option in `clingo` and `xclingo`. For such command, s(CASP) obtains the minimal s(CASP) justification trees shown in Figure 6.2.

It must be mentioned that the exact output of s(CASP) has been summarized for brevity: the partial models and the particular bindings for the variables that accompany the justifications have been removed from this and any subsequent output. When comparing these justifications to other ASP explainability tools like `xclingo`, the reader must recall that they are not computed using either answer sets or partial answer sets as a starting point, but they come naturally from the top-down evaluation that s(CASP) performs to answer the queries. Although they do relate to a corresponding partial answer set, one cannot (directly) tell whether two justifications belong to the same model or not. In other words, one cannot tell whether both explanations refer to the same *world* (or *solution*) and so, are simultaneously valid or not. To know that for a group of justifications, it would be necessary to check if their corresponding partial models and the *global constraints* justification that is being omitted here are compatible. For instance, it is easy to see how each of the 7 justifications from s(CASP) relate to each one of the explanations obtained by `xclingo` for the three answer sets, with the exception of Justification 6 (in Output 6.2), that corresponds with both explanations of `sentence(clare, innocent)` in Answer 3 (in Output 4.7) In contrast, in Output 4.7 obtained from `xclingo`, alternative explanations corresponding to the same solution are shown together (like in Answer 3 of Output 4.7).

Except for the justifications concerning `clare`, the appearance and the structure of both systems' tree justifications/explanations resemble almost the same. A difference is that, when justifying that a person drove drunk (respectively not drunk) the fact that her alcohol level is above the threshold (respectively below) is included. But, by far, the greatest difference is that s(CASP) includes the explanation of the default negated literal `not punish(clare)`, to justify `sentence(clare, innocent)`. In justifications 5, 6 and 7, `clare` is justified to be `innocent` because (summarizing) she did not resist authority and she was not drunk while driving (namely, her alcohol level was lower than 30).

Number of generated explanations

By executing Program 4.6, introduced in Section 4.2.5, as it happens with support graphs and `xclingo`, this leads to an exponential number of explanations (that is, the worst case) when explaining any `signal/1` atom. For instance, by querying the atom `signal(5)`, s(CASP) obtains a total of $2^5 = 32$ different explanations. This is also the number of causal graphs and *Tree Explanations* generated by Fandinno et al.'s and Erdem et al.'s approaches respectively. Output 6.3 shows one of the 32 explanations obtained.

Another important similarity with respect to our approach is that, within the same explanation, the reasons for each atom are fixed. That is, if an atom appears as a cause several times within an ex-

```
% --------------- Answer 1 (0.000 sec)
   query ←
      sentence(gabriel,prison) ←
         punish(gabriel) ←
            drive(gabriel) ∧
            alcohol(gabriel,40) ←
               w1 ←
            40>30 ∧
            person(gabriel)

% --------------- Answer 2 (0.000 sec)
   query ←
      sentence(gabriel,prison) ←
         punish(gabriel) ←
            drive(gabriel) ∧
            alcohol(gabriel,40) ←
               w3 ←
            40>30 ∧
            person(gabriel)

% --------------- Answer 3 (0.000 sec)
   query ←
      sentence(gabriel,prison) ←
         punish(gabriel) ←
            resist(gabriel) ←
            person(gabriel)

% --------------- Answer 4 (0.000 sec)
   query ←
      sentence(gabriel,prison) ←
         punish(gabriel) ←
            resist(gabriel) ←
               w3 ←
            person(gabriel)

% --------------- Answer 5 (0.000 sec)
   query ←
      sentence(clare,innocent) ←
         person(clare) ∧
         not punish(clare) ←
            drive(clare) ∧
            not alcohol(clare,A | {A ∉ [5]}) ∧
            proved(drive(clare)) ∧
            alcohol(clare,5) ∧
            5=\30 ∧
            not resist(clare)

% --------------- Answer 6 (0.000 sec)
   query ←
      sentence(clare,innocent) ←
         person(clare) ∧
         not punish(clare) ←
            drive(clare) ∧
            not alcohol(clare,A | {A ∉ [5]}) ∧
            proved(drive(clare)) ∧
            alcohol(clare,5) ∧
            5=<30 ∧
            not resist(clare)

% --------------- Answer 7 (0.000 sec)
   query ←
      sentence(clare,innocent) ←
         person(clare) ∧
         not punish(clare) ←
            drive(clare) ∧
            not alcohol(clare,A | {A ∉ [5]}) ∧
            proved(drive(clare)) ∧
            alcohol(clare,5) ∧
            5=<30 ∧
            not resist(clare)
```

Output 6.2: Explanatory output obtained by s(CASP) for Program 4.1 after replacing the choice rule by the loop in Figure 6.7.The output was obtained by executing Command 6.1. The justifications explain the query `sentence(P,S)`.

```
query ←
    signal(5) ←
      fire_b(4) ←
          signal(4) ←
            fire_b(3) ←
                signal(3) ←
                  fire_b(2) ←
                      signal(2) ←
                        fire_b(1) ←
                            signal(1) ←
                              fire_b(0) ←
                                  signal(0).
% Model
{ fire_b(0),          fire_b(2),      fire_b(4),    signal(1),        signal(3),      signal(5),
  fire_b(1),          fire_b(3),      signal(0),    signal(2),        signal(4)
}
```

Output 6.3: One of the 32 explanations from the output of s(CASP) for Program 4.6. The query is `signal(5)`.

planation, its explanation (the *subtree* hanging from it) is the same in all its appearances. For instance, consider again Program 6.1. In the case of s(CASP) it generates only two explanations where the cause for both `up(a)` and `up(b)` is the same `wireless` (or s, or y). This coincides with the number of support graphs (and therefore with the explantions obtained via `xclingo`), the $\subseteq$ −minimal causal graphs from Fandinno's and the shortest explanation trees obtained by Erdem et al.

Explanation Design with s(CASP)

In terms of explanation design, s(CASP) provides tools for controlling the filter in which information is present in the justifications and also ways to generate natural language explanations.

Concerning filtering, the options that s(CASP) provides mainly focus on controlling which negated literals should be justified. To that aim, the user can make use of several command line options (explained in [7] and [10]), including:

- `short`: shows the negated literals selected with `#show` directives.
- `mid`: adds the rest of user-defined predicates (positive and/or classically negated).
- `long`: generates the complete s(CASP) justification tree, including auxiliary predicates.

In newer versions of s(CASP) (with respect to [10]), there also exists the option `pos` which removes negated literals from the tree, but preserves the positive par of the subgraph hanging from them. For instance, the execution of Command 6.2, makes Justification 5 (and thus, also 6 and 7) become what we can see in Output 6.4.

```
scasp -s1 -pos -tree dont_drive_drunk_choice.lp -query="sentence(clare, S)"
```
(Command 6.2)

Although not detailed, the result of this deletion resembles a *forgetting* operation (defined in Definition 10) over the nodes of the tree that correspond to default negated literals. Indeed, the existence of the `long` option reveals that this is done by default to hide all the internal s(CASP) predicates.

Output 6.4: Output for Program 4.1 after replacing the choice rule by the loop in Figure 6.7. The output was obtained by calling Command 6.2.

However, selecting a particular set of literals is not possible for now, which can become a problem when dealing with causal domains and representations, as we argue in section 4.2.4. For instance, the justification shown in Output 6.4 could not be considered causally correct since being a person is not a cause of being innocent. To be fair, in this particular case the literal `person(P)` could be removed from any rule in Program 4.1 since s(CASP) does not need to ensure the safety of the variables, thus fixing the issue. However, as it is explained in Section 4.2.4, the use of extensional rules as

```
person(S) :- student(S).
```

would still represent an issue.

Moreover, the system offers an automatic way to generate natural language explanations, like the one shown in Output 6.5, through the use of the `-human` option.

```
scasp -s1 -tree -human dont_drive_drunk_choice.lp -query="sentence(gabriel, S)"
```
(Command 6.3)

```
sentence holds for clare, and innocent, because
    there is no evidence that punish holds for clare, because
        there is no evidence that drove_drunk holds for clare, because
            drive holds for clare, and
            there is no evidence that alcohol holds for clare, and A not equal to 5, and
            drive holds for clare, justified above, and
            alcohol holds for clare, and 5, and
            5 is less than or equal to 30
        there is no evidence that resist holds for clare
```

Output 6.5: Output for Program 4.1 after replacing the choice rule by the loop in Figure 6.7. The output was obtained by calling Command 6.3.

The explanation preserves the same indentation structure, replacing $\leftarrow$ and $\wedge$ symbols with the words `because` and `and` respectively. Besides, it collects the values from the literals and constructs natural language expressions that capture the original meaning of the replaced literals by using predefined patterns like *there is no evidence that <predicate name> holds for <value>*. Except for the case that user-unfriendly, cumbersome predicate names written by the KR engineer could be shown in the justification, this option is a very effective way of making automatic technical explanations more accessible without expending any effort in explanation design. It could also be highly useful during the explanation design process itself. Throughout the explanation prototyping phase, these types of explanations could be presented to the user to gather feedback and thus involve the user more easily in the design process.

Finally, s(CASP) also gives the option to design the text templates that replace the literals in the justification. This is done by the use of `#pred` directives, that target certain predicates and define a custom text very similarly to how `trace` annotations work. For instance, Figure 6.8 shows some `#pred` for Program 4.1.

```
#pred drive(P) :: '@(P) has driven'.
#pred alcohol(P, A) :: '@(P)\'s alcohol level is @(A)'.
#pred punish(P) :: '@(P) is punished'.
#pred resist(P) :: '@(P) resisted'.
#pred sentence(P,S) :: '@(P:person)\'s sentence is @(S)'.
```

Figure 6.8: Some `#pred` directives that associate some custom text to particular predicates.

```
the person clare's sentence is innocent, because
   clare has driven, and
   clare has driven, justified above, and
   clare's alcohol level is 5, and
   5 is less than or equal to 30
```

Output 6.6: Output for Program 4.1 together with `#pred` directives in Figure 6.8 after using Command 6.4.

These clauses are meant to be kept in a separate file, and the `(Variable:Name)` marks are used to link the variable value to the text, where `Name` is used in the natural language patterns to build more meaningful texts. Output 6.6 shows the effect of adding the directives as in Command 6.4.

```
scasp -s1 -pos -human -tree dont_drive_drunk.lp dont_drive_drunk.pred -query="sentence(clare, S)"
```
(Command 6.4)

If compared with `trace` and `trace_rule` annotations, `#pred` directives provide less flexibility. They operate solely at the predicate level, but not at the rule or atom (or literal) level. That is, either we annotate all the literals belonging to a certain predicate or we annotate none. It is not possible to restrict a custom text to the atoms derived from particular rules, nor to a subset of the literals based on the values of their variables. This means that, except for predicates that appear only as the head of one rule, the `trace_rule` annotation cannot be replicated. Even for `trace` annotations, some of the original text annotations we wrote in the first examples (see Program 4.2) would need the introduction of auxiliary artifacts to be replicated, for instance, those in Figure 6.9.

```
        %!trace {punish(P), "% drove drunk", P} :- punish(P), resisted(P).
        %!trace {punish(P), "% resisted authority",P} :- drive(P), alcohol(P, A), A > 30
```

Figure 6.9: Some `xclingo` annotations that cannot be replicated in s(CASP).

Interestingly, s(CASP) exhibits an also powerful feature that allows to generate a natural language-based translation of the ASP code. This is related to Sartor et al.'s *Logical English* [64], for which there is indeed an integration with s(CASP) [91]. The translation works similarly to the natural language generation for justifications and uses similar patterns. Having such a tool for a declarative language like ASP is of great importance, as it can help bridge the gap between the user world and the technical world Beyond the idea that the `#pred` directive or `xclingo` text annotations can be used to sketch a meaningful description of the code and directly provide a natural language explanation of the specification itself. That is, aiming not only to achieve accountable explanations but also to provide accountable specifications.

6.4 Offline Justifications

Offline Justifications

The approach followed by Pontelli et al. [84, 85], develops a notion of justification of an atom under the ASP semantics. The proposed justifications take the form of *labeled directed graphs* that explain the truth value of the atom (either true or false) with respect to an answer set and a set of assumptions. The authors first define a more general notion of these structures called *Offline Explanation Graph* and then specialize it to the concept of *Offline Justification*. These are two notions of graphs, the former allowing positive cycles whereas the latter no. It is important to remark that the definition firstly provided in [85] was later updated in [100], where similar concepts were devised, but under the names of *Derivaion Path* and *Explanation Graphs* respectively.

We start revising the notion of the explanation graph. Such graphs are defined with respect to a program P, an answer set A, a set of atoms assumed to be false U and the atom to be explained a. A vertex can be one of the following:

- A positive atom p labeled with $+$ (from now on written p^+). In other words, atoms that are true w.r.t are the concerning answer set (i.e. $p \in A$).
- A (default) negative atom labeled with $-$ (from now on written p^-). That is, atoms that are false w.r.t the concerning answer set (i.e. $p \notin A$).
- Either one of the following three: $\top$, $\bot$, or `assume`.

Intuitively, $\top$ is used as a cause for facts, $\bot$ is used as a cause for negative atoms (i.e. are not in A) so there is no rule in P having such an atom in its head, and `assume` is used to give a cause for atoms in U (i.e. those assumed as negative). Let the graph edges are tuples $\langle x, y, s\rangle$, where x is the source vertex, y is the target vertex, and $s \in \{+, -, \circ\}$ is the edge label. For each positive atom vertex x^+, (only) one rule $r \in P$ supported by A is selected to draw edges $\langle x^+, y^+, +\rangle$ for each positive literal y in the body of r; and $\langle x^+, y^-, -\rangle$ for each positive literal "not y" in the body of r. For each negative atom vertex x^-, (only) one rule $r \in P$ not supported by A is selected to draw edges $\langle x^-, y^+, -\rangle$ for each positive literal y in the body of r; and $\langle x^-, y^-, +\rangle$ for each positive literal "not y" in the body of r. For each atom $u \in U$, an edge $\langle u^-, \text{assume}, \circ\rangle$ is drawn. Finally, cycles are allowed over negative atoms but not over positive atoms.

Reconsider now the following example Program 6.4 which modifies the previously shown Program 6.1 (see Section 6.1), by introducing abnormal conditions that may prevent the bomb from exploding (see Figure 6.4). We include these new rules for illustrating the usage of default negation and assumptions U for this approach.

$$\begin{array}{rll}
open & \leftarrow up(a), up(b), not\ abnormal & \\
abnormal & \leftarrow wet & \qquad abnormal \leftarrow battery \\
wet & \leftarrow not\ battery & \qquad battery \leftarrow not\ wet
\end{array}$$

Program 6.4: Some modifications applied to Program 6.1, now the bomb may not explode if it is wet or if the battery fails.

For building the explanation graphs for atom *bomb*, a set of assumptions U is needed first. This set of assumptions is computed with respect to to a corresponding answer set A. Program 6.4 has two answer sets in neither of which the bomb explodes, in particular a first answer set such that $wet \in A_1$, where the wetness prevents the explosion and a second answer set such that $battery \in A_2$ where the battery fails. In the case of answer set $A_1 = \{wet, wireless, up(a), up(b), abnormal\}$ the only valid set of assumptions is $U = \{battery\}$, for which we obtain the explanation graphs in Figure 6.10.

$$\begin{tikzcd}
 & bomb^- \arrow[d, "+"] & \\
 & open^- \arrow[dl, "+"] \arrow[d, "+"] \arrow[dr, "-"] & \\
up(a)^+ \arrow[d, "+"] & up(b)^+ \arrow[dl, "+"] & abnormal^+ \arrow[d, "+"] \\
(s)\,wireless^+ \arrow[d, "+"] & & wet^+ \arrow[d, "-"] \\
\top & & battery \arrow[d, "\circ"] \\
 & & assume
\end{tikzcd}
\qquad
\begin{tikzcd}
 & bomb^- \arrow[d, "+"] & \\
 & open^- \arrow[dl, "+"] \arrow[d, "+"] \arrow[dr, "-"] & \\
up(a)^+ \arrow[d, "+"] & up(b)^+ \arrow[dl, "+"] & abnormal^+ \arrow[d, "+"] \\
(y)\,wireless^+ \arrow[d, "+"] & & wet^+ \arrow[d, "-"] \\
\top & & battery \arrow[d, "\circ"] \\
 & & assume
\end{tikzcd}$$

EG₁　　　　*EG₂*

Figure 6.10: The two explanation graphs drawn according the definitions in [100], when the assumption set is $U = \{battery\}$. Another two could be obtained when the assumption set is $\{wet\}$.

Note the labels (s) and (y) for the $wireless$ atoms in the graphs. Although in the context of this approach, rules are not labeled we include them for clarity when comparing the graphs with the previously reviewed approaches.

The graph EG_1 explains that atom *bomb* is false (i.e. is labeled with $-$) because it has a positive dependency towards atom *open* which is negative. Then continues justifying that, although positive dependencies with $up(a)$ and $up(b)$, which are both true, are met, the negative dependency with $abnormal$ which is not met since the atom is true. Then, $abnormal$ is true because wet is true, because $battery$ is assumed to be false. Finally, $up(a)$ and $up(b)$ are both true because they have a positive dependency with $wireless$ which is true because it is a fact. The only difference w.r.t EG_2 is that the rule labeled with (y) is used to support the atom $wireless$ instead of the rule labeled with (s).

The amount of explanations found for atoms $up(a)$ and $up(b)$ relate to Fandinno's causal graphs (Figure 6.1) and Erdem et al. explanation trees (Figure 6.5) in the same way that our approach's support graphs (Figure 6.3). This is because both approaches fix the rule for deriving atom $wireless$ in their definitions for the graphs, whereas the other two allow explaining the same atom with different rules on different occurrences. In the case of positive programs, the explanations produced by this approach and support graphs should coincide.

xASP2 System

Trien, Son, Pontelli and Balduccini [100] provide a tool called `exp(ASP)` that is inspired by offline justifications. Later in the same year, they provided an incremental version of this tool called `exp(ASP`^{c}`)` [101] which includes explanations for choice rules and aggregates.

One year later, Trien, Son and Balduccini [102], present a new tool called `xASP` which solves some shortcomings of the previous tool. In the latest two publications [5, 6] a new version of this tool called `xASP2` is presented. This version includes explanations for new language extensions like constraints and also implements the explanation computation with a method involving the use of ASP meta-programming. That is, like `xclingo`'s *ad-hoc* explainer program (see Section 4.3), it is a generated ASP program that computes the explanations for the original ASP program. In particular, the meta-program used by `xASP2` follows the approach described in [22].

As a tool, recall `xASP2` is not an implementation of offline justifications since the explanations obtained do not correspond to that definition, although it takes some inspiration for some aspects such as the assumptions. The tool is provided as a Python package, publicly published on github [1]. Given a program, a model and a set of atoms to explain, the tool provides an API with several endpoints to compute (i) the minimal assumption set, (ii) the explanation sequences; and (iii) the explanation graphs. Figure 6.11 illustrates the use of the Python API.

The minimal assumption sets and both the explanation DAGs and sequences are provided in the form of ASP atoms representing the assumptions or the edges of the graphs respectively. The generated `igraph` corresponds to a generated HTML depicts page that visually represents the graph and provides some functionality like doing zoom-in or zoom-out on the graphs.

Consider the program *dont_drive_drunk.lp* again (see Program 4.1 in Section 4.2.1). Figure 6.12 shows the explanations obtained by `xASP` for the atoms `sentence(clare, innocent)` (left) and `sentence(gabriel, prison)` (right). The explanations coincide with those obtained by `xclingo` with the only difference of the inclusion here of the negative literal `not punish(P)` from rule in line 12 (of Program 4.1 in Section 4.2.1). However, as offline justifications or `s(CASP)`, the tool can also produce explanations for atoms that are not true with respect to to the corresponding model. For instance, Figure 6.13 shows the explanations for *not bomb* in Program 6.4 both with `xASP` and `s(CASP)`.

Unlike its corresponding explanation graph EG_2 from Figure 6.10 and the explanation obtained by `s(CASP)`, `xASP` just uses the literal preventing *open* to be derived (i.e. $abnormal$ is true) to explain that *bomb* is false. Note also that `s(CASP)` generates two explanations for answer set A_1 corresponding to the use of each of the two $wireless$ rules for justifying atoms $up(1)$ and $up(2)$.

We will test now the explanations obtained by `xASP` for aggregates. To this aim, we will refer back to Program 4.8. Since the version of the tool being reviewed in this dissertation does not support recursion in aggregates, we need to rewrite the rule containing the `#sum` aggregate (see Program 6.5).

Each set in the original aggregate has been isolated in a rule deriving a common predicate `__aggregate_set/2` that is later used in line 2 to collect all the atoms participating in the aggregate. Reasonable values for variable `Sum` need to be provided as value invention is not supported. Please take this difference into account when comparing the output from `xclingo` in Output 4.13 against `xASP`'s in Output 6.14

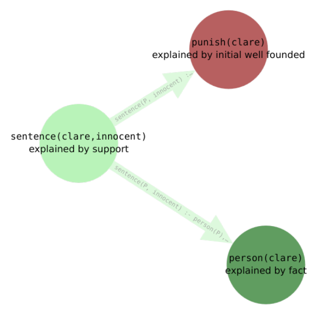
[1] https://github.com/alviano/xasp

```python
from xasp.entities import Explain
from dumbo_asp.primitives import Model

with open("dont_drive_drunk.lp", "r") as rfile:
    prog = rfile.read()

explain = Explain.the_program(
    prog,
    the_answer_set=Model.of_program(prog),
    the_atoms_to_explain=Model.of_atoms("sentence(gabriel, prison)"),
)

# explain.compute_minimal_assumption_set()  # for computing the minimal assumption set
# print(explain.minimal_assumption_set())

# explain.compute_explanation_dag()  # for obtaining the graphs
# print(explain.explanation_dag())

# explain.compute_explanation_sequence()  # for obtaining the explantion sequence
# print(explain.explanation_sequence())

explain.compute_igraph()  # for generating a html version of the graphs
print(explain.show_navigator_graph())
```

Figure 6.11: Python snippet of code for using xASP.

```
polluted_river(River) :-
    river(River),
    #sum{P: __aggregate_set(P,River)}=Sum,
    Sum = 51..1000.

    __aggregate_set(P1,River) :- entity(E, T), chemical(E, River, P1).
    __aggregate_set(P2,River) :- river(River), entity(E, T), nutrient(E, rin, P2).
```

Program 6.5: Modification of Program 4.8. The rule containing the aggregate is rewritten for being compatible with xASP.



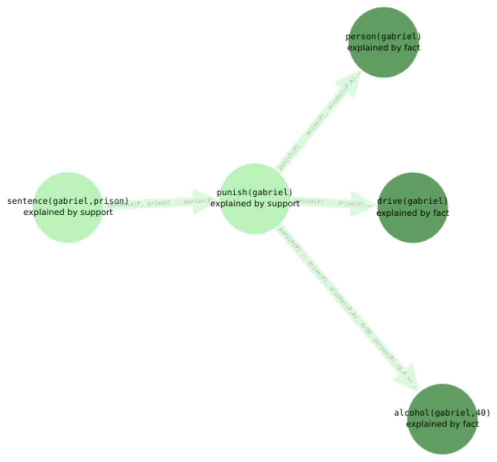
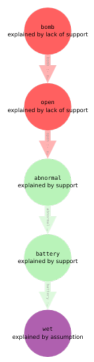


Figure 6.12: Explanations obtained with xASP for Program 4.1. The explanation for `sentence(clare, innocent)` is shown on the left side of the figure whereas the explanation for `sentence(gabriel, prison)` is on the right side.



```
query <*$\leftarrow$*>
  not bomb <*$\leftarrow$*>
      not open <*$\leftarrow$*>
        up(1) <*$\leftarrow$*>
            wireless <*$\land$*>
        up(2) <*$\leftarrow$*>
            proved(wireless) <*$\land$*>
        abnormal <*$\leftarrow$*>
            wet <*$\leftarrow$*>
              not battery <*$\leftarrow$*>
                  chs(wet).
```

Figure 6.13: Explanation obtained with xASP for Program 6.4 at the the left and by `s(CASP) at the right`. Query is *not bomb*. The answer set being explained is A_1 such that $wet \in A_1$.

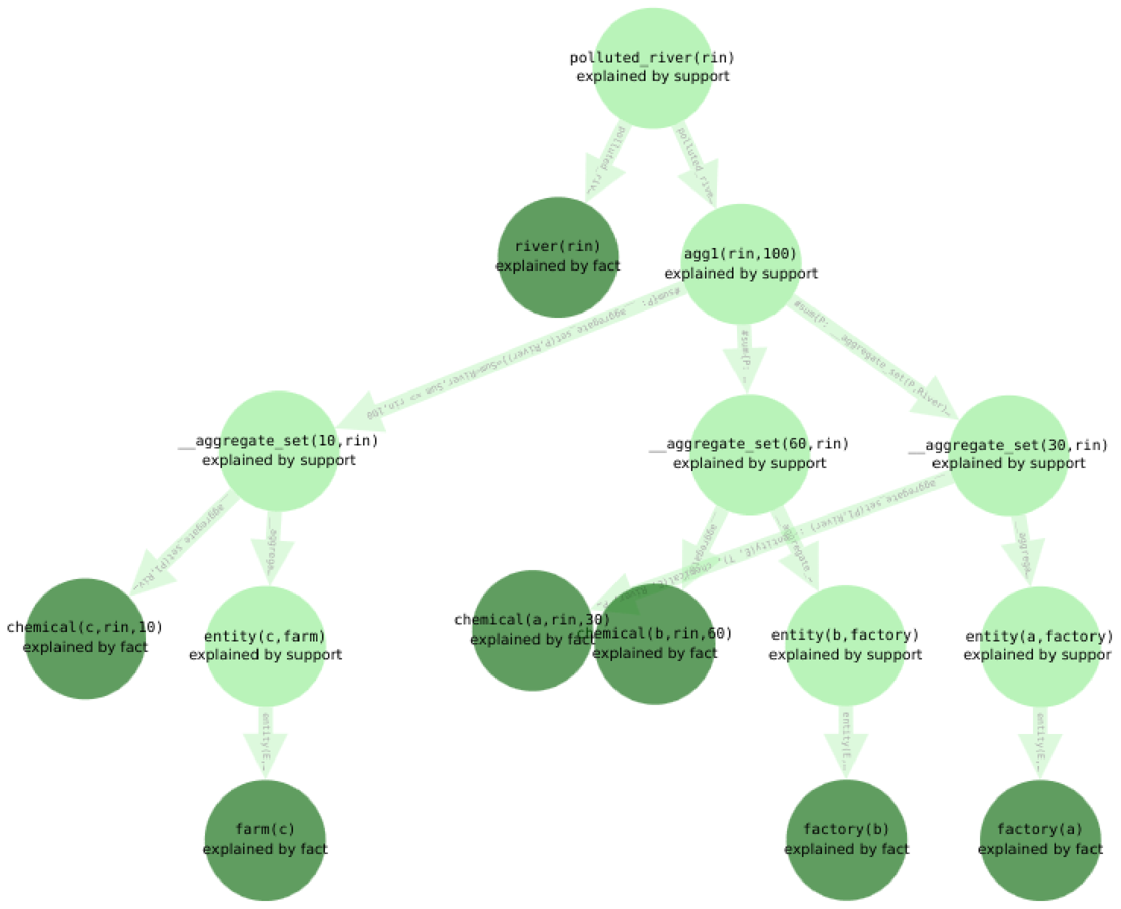


Figure 6.14: Explanation from xASP obtained for Program 6.5.

Although the structure of the explanation is not the same, we can see how, as in `xclingo`, all atoms are used instead of including only a sufficient set.

Explanation Design with xASP

System x(ASP) is mainly thought for debugging in ASP, producing fully detailed technical explanations. Some effort is made by providing an interactive environment where the graphs are depicted in a browser. However, without any means of selecting particular information of interest for the user or expressing the information in natural laguage means, the system is not able to produce accountable or commonsense explanations.

6.5 m-justifications And r-justifications For C-Atoms

The recent approach by Eiter, Geibinger and Detsch [38, 39] proposes the use of *Abstract Constraint Atoms* (c-atoms) [69] for extracting two different types of explanations. In particular, they propose two different kinds of explanations, namely *m-justifications*, which explain the truth value of an atom with respect to a given model and *r-justifications* that proceed in a top-down fashion and take the whole program into account as a chain of activated rules.

Their notions of explanations rely on programs containing c-atoms. With respect to a c-literal L, intuitively, an m-justification for L with respect to to a total model I as the smallest partial justification J such that $J \leq I$ for which L is true. In other words, the smallest set of sufficient (positive and negative) atoms in I needed to make L hold.

Consider the following aggregate from Example 2 of [38],

#sum {2: a, 1: b; 1 : c} > 1.

that can be represented as the c-atom $CA_1 = \langle D_1, C_1 \rangle$ where $D_1 = \{a, b, c\}$ and $C_1 = \{\{a\}, \{b, c\}, \{a, b\}, \{a, c\}, \{a, b, c\}\}$ are respectively the *domain* and the *satifiers* of CA_1 Then, following the definition, there exist two partial interpretations for CA_1, namely $\langle \{a\}, \emptyset \rangle$ meaning that atom a alone is sufficient for the aggregate to hold, and $\langle \{b, c\}, \emptyset \rangle$ meaning that atoms b and c are (together) sufficient as well. The only m-justification for $\overline{CA_1} = \langle D_1, \{\emptyset, \{b\}, \{c\}\} \rangle$, that is CA_1 being false, is the partial interpretation $\langle \emptyset, \{a, c\} \rangle$ meaning that, in the case a and c are false, that is sufficient to know that the aggregate will not hold.

This notion is of great importance for the discussion on how to explain aggregates. Previously in Section 4.2.6 we introduced how `xclingo` explains aggregates. Also there, we commented on how the notion of the sufficient causes of an aggregate could be useful in some cases. The particular implementation for achieving those explanations is explained in Section 4.3.2. The sufficient causes found by this approach coincide with the example given in Section 4.2.6. In particular, for Program 4.8 m-justifications for the aggregate checking if river `rin` is polluted are

$$\langle \{ \mathit{chemical}(b, \mathit{rin}, \mathit{60}) \}, \emptyset \rangle$$

$$\langle \{ \mathit{chemical}(a, \mathit{rin}, \mathit{30}), \mathit{nutrient}(b, \mathit{rin}, \mathit{30}) \}, \emptyset \rangle$$

At the end of Section 6.4, we compare the output of `xclingo` and the `xASP` system regarding the explanation of aggregates. Both systems gather all the participating atoms in the aggregate and incorporate them into the explanations. However, it appears that current explainability ASP systems are missing a feature for extracting the sufficient causes of an aggregate. The concept of m-justification could provide guidance for future implementations on how explainability systems should handle aggregate explanations.

6.6 LABAS justifications

Assumption Based Argumentation (ABA) framework [15, 36] is a computational framework for default reasoning. The explanability approach by Schulz and Toni, called *LABAS justification* [93, 94, 95], leverages the notions of *arguments* and *attacking trees* from ABA frameworks. LABAS explains the truth value of a literal for a given answer set and program. The original definitions are provided with respect to the translation of a logic program into an ABA framework. Due to the correspondence between a logic program and its corresponding translation [96], in the survey [48], Schulz and Fandinno provide the correspondent definitions with respect to the original logic program. We will use the definitions provided by the latter here.

We start from the (very general) notion of an *argument* for a literal l with respect to program P. Intuitively, the argument of l is a finite tree where l corresponds to the root node and the rest are literals from the program so that: (1) children of non-leaf nodes corresponding to positive literals h, are the literals of a rule $r \in P$ such that h is the head of r; and (2) fact literals and negative literals are always leaf nodes. That is, an argument represents a derivation for a literal stopping only when negative literals and facts are reached. Figure 6.15 shows arguments (displayed as a tree) for *bomb* and $abnormal$ literals with respect to Program 6.4.

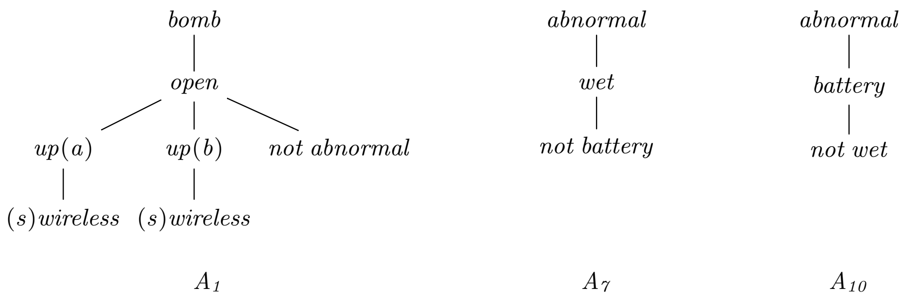


Figure 6.15: ABA arguments for literals *bomb* and $abnormal$ from Program 6.4.

Note that there exist other 3 arguments like A_1 but replacing any $(s)wireless$ node by $(y)wireless$. We will consider just A_1 for the examples although we will clarify what would happen if we consider the four arguments. Note that other arguments for other literals such as for instance *open* are a subtree of one of those subtrees.

Arguments can also be denoted as $A : (AP, FP) \vdash l$ where A is a unique name for the argument, AP is the set of assumption premises (i.e. the set of negated (leaf) literals in the argument), FP is the set of fact premises (i.e. the set of positive (leaf) literals in the argument) and l is the literal the arguments refer to. For instance, argument A_1 can be denoted as $A_1 : (\{not\ abnormal\}, \{(s)wireless\}) \vdash bomb$. The rest of the arguments that are subtrees of those in Figure 6.15 are listed Figure 6.16.

$A_1 : (\{not\ abnormal\}, \{(s)wireless\}) \vdash bomb$	$A_7 : (\{not\ battery\}, \{\}) \vdash abnormal$
$A_2 : (\{not\ abnormal\}, \{(s)wireless\}) \vdash open$	$A_8 : (\{not\ battery\}, \{\}) \vdash wet$
$A_3 : (\{\}, \{(s)wireless\}) \vdash up(1)$	$A_9 : (\{not\ battery\}, \{\}) \vdash not\ battery$
$A_4 : (\{\}, \{(s)wireless\}) \vdash up(2)$	$A_{10} : (\{not\ wet\}, \{\}) \vdash abnormal$
$A_5 : (\{\}, \{(s)wireless\}) \vdash (s)wireless$	$A_{11} : (\{not\ wet\}, \{\}) \vdash battery$
$A_6 : (\{not\ abnormal\}, \{\}) \vdash not\ abnormal$	$A_{12} : (\{not\ wet\}, \{\}) \vdash not\ wet$

Figure 6.16: Abbreviated ABA arguments for literals in Program 6.15.

Although each argument possesses its own tree form and abbreviated denotation, for the sake of this explanation we can group the arguments by from which tree argument of Figure 6.15 they can be obtained from by noticing those that share assumption and fact premises. In this particular case, arguments from A_1 to A_6 come from the tree form of A_1, A_7 to A_9 come from the tree form of A_7 and A_{10} to A_{12} come from the tree form of A_{10}.

If we now consider the previously mentioned variations of argument A_1 we would obtain the following two in abbreviated form: $A_{13} : (\{not\ abnormal\}, \{(y)wireless\}) \vdash bomb$ and $A_{14} : (\{not\ abnormal\}, \{(s)wireless, (y)wireless\}) \vdash bomb$. Although there are a total of four variations, the two of them resulting in flipping the occurrences of $(s)wireless$ and $(y)wireless$ result in the same abbreviated form as AP and FP are set. This means that we would have three ways to explain the literal *bomb* being true in an answer set. Note that this number of explanations coincides with the number of proof graphs in Figure 6.2, and that the corresponding support graphs 6.3 are a subset of both of them.

Now the notion of attacking trees from ABA frameworks is used to define *attack trees justifications*. By defintion, an argument $A_a : (AP_a, FP_a) \vdash l_a$ attacks another argument $A_d : (AP_d, FP_d) \vdash l_d$ if $not\ l_a \in AP_d$. Intuitively, an attack tree justification for a literal l w.r.t an answer set M, is a tree of attacking arguments such that the argument in the root of the tree is of the form $A : (AP, FP) \vdash l$. Each argument is labeled $+$ or $-$ if the literal they refer to is (respectively) in M or not. In the case of arguments labeled with $+$ all the literals in its AP must hold with respect to the model, if that is not the case the argument cannot be used. Arguments labeled with $-$ are attacked by only another argument, thus meaning that to prevent the derivation of an atom only one of the premises needs to be false. Arguments labeled with $+$ are attacked by an argument for each of their assumption premises, thus meaning that for proving the derivation of an atom all its premises must be true. Figure 6.17 shows an attacking tree justification for literal *bomb* with respect to Program 6.4 and answer set $A_1 = \{wet, wireless, up(a), up(b), abnormal\}$.

The attack tree justification ATJ_1 infinitely repeats itself once it reaches the loop between wet and $battery$. Note that we cannot $A_{10} : (\{not\ wet\}, \{\}) \vdash abnormal$ to attack A_1 since $not\ wet$ does not hold for A_1 (i.e. $wet \in A_1$). Recall that this condition only applies for arguments labeled with $+$, that is why A_1 can be used even though $abnormal \in A_1$ Intuitively we can read the justification as:

1. *bomb* is false since, as A_1 (which is false) says, it negatively depends on $abnormal$, which is true.

2. This latter fact is argumented by A_7 (which is true), saying that $abnormal$ is true because $battery$ does not hold.

$$\begin{tikzcd}
A_1^- : (\{not\ abnormal\}, \{(s)wireless\}) \vdash bomb \\
A_7^+ : (\{not\ battery\}, \{\}) \vdash abnormal \arrow[u] \\
A_{11}^- : (\{not\ wet\}, \{\}) \vdash battery \arrow[u] \\
A_8^+ : (\{not\ battery\}, \{\}) \vdash wet \arrow[u] \\
A_{11}^- : (\{not\ wet\}, \{\}) \vdash battery \arrow[u] \\
A_8^+ : (\{not\ battery\}, \{\}) \vdash wet \arrow[u] \\
\vdots \arrow[u, two heads] \\
ATJ_1
\end{tikzcd}$$

Figure 6.17: Attacking tree justification for $bomb$ with respect to Program 6.4 and answer set A_1.

3. Which is supported by A_{11} saying that for deriving $battery$ we would have needed wet to be false.

4. But as A_8 counterarguments, wet is true because $battery$ is false.

5. From this point on the tree infinitely repeats A_{11} and A_8 attacking each other, reflecting the loop between both atoms.

Finally, a *LABAS justification* can be obtained from an attack tree justification in a two-step process. Intuitively, support and attack relations are obtained from the tree. These are pairs representing the directed edges of a graph. The support edges go from the assumptions of an argument (i.e. AP and FP) to the literal that the argument refers to. For the arguments that are not The attack edges go from the arguments (i.e. the literals concluded by the arguments) to the assumptions they attack. All literals (including those from assumptions of the arguments) are labeled with $+$ or $-$ if they are in (or respectively not in) the model corresponding model M. Edges are also labeled with the same label as its source node.

Figure 6.18 shows the LABAS justification obtained from attacking tree ATJ_1 in Figure 6.17.

Dashed edges represent support relations whereas solid edges represent attacking relations. Note that, since the graph is drawn from a set (of edges) it is now finite. Another important point is that the LABAS justification of an atom that is false w.r.t the answer set (as it is our case with $bomb$) is defined as the set of all the possible LABAS justifications for that atom. This follows the idea that for an atom to be false all the ways to derive such an atom have to be unsuccessful. In our case, LJ_1 corresponds to only one way of deriving a bomb that was falsified. To complete the justification, the rest of LABAS justifications using arguments A_{12} and A_{13} would have to be included.

As conclusion, LABAS justifications explain the truth value of literals in an answer set with respect to a program in terms of the negative literals used in the rules of the program. They focus on how the

$$\begin{tikzcd}
bomb^{-}_{A_1} \\
not\ abnormal^{-}_{asm} \arrow[u, dash, dashed, "-"] \\
abnormal^{+}_{A_7} \arrow[u, "+"] \\
not\ battery^{+}_{asm} \arrow[u, dash, dashed, "+"] \\
battery^{-}_{A_{11}} \arrow[u, "-"] \\
not\ wet^{-}_{asm} \arrow[u, dash, dashed, "-"] \\
wet^{+}_{A_8} \arrow[u, "+"] \\
LJ_1
\end{tikzcd}$$

Figure 6.18: LABAS justification for $bomb$ with respect to Program 6.4 and answer set A_1. In particular, it comes from attacking tree ATJ_1 in Figure 6.17.

different default negated atoms enable or prevent the derivation of other atoms. The positive parts of the derivations are given less importance. Indeed, they are collapsed into layers where only the facts are shown. In contrast with offline justifications that explain negative literals as assumptions with respect to a corresponding answer set, LABAS detailedly explains the different ways in which the default negated literals of the program may intervene in the derivations.

6.7 Justification Theory

Justification Theory initially developed in [33] and further developed more recently in [14, 32, 71] is an abstract formalism for justifications. This theory defines a notion called *Justification Frame* which is a structure $JF_P = (F, L, P)$ obtained from a logic program P. Where L is the set of all possible literals over the signature of program P and F a set of facts named *fact space*, and additionally they denote a set $F_\circ = F \setminus L$ and named *parameter facts*. One of the intuitions for a justification frame is it can be understood as a casual system where literals L are endogenous facts governed by the rules in P and parameter facts $F_\circ$ are exogenous facts governed by an agent or system. In this sense, as support graphs and `xclingo`, justification frames envision a logic program as a set of causal relations over a set of events that are the atoms occurring in the program

Thanks to 4-valued interpretations and to the definition of an abstract derivation operator, justification frames can be defined over different semantics. This allows the authors to compare logic programs interpreted under different semantics in terms of their corresponding justification frames.

From justification frames, they define *JF-justifications* as a subset of rules $R \subseteq P$ containing rules $r \in R$ such that the head of r are $x \in L$. Additionally, a JF-justification can only contain at most one rule for each $x \in L$. This coincides with Definition 6 of support graphs where only one rule supporting a particular atom can be used to build the graph. A JF-justification JFJ is called *complete* if $\forall x \in L, \exists r \in JFJ$ such that $head(r) = x$. Then they provide a correspondence between a complete JF-justificaton JFJ and a directed graph whose leaves are parameter facts from $F_\circ$ and the children of non-leaf x are $S = \{S | (r \leftarrow S) \in JFJ \wedge head(r) = x\}$, or in words: the literals occurring in the body their corresponding rule $r \in JFJ$ Although it has not been properly studied yet, it seems that support graphs and these graphs could be similar at least for the atoms corresponding to a particular model of the program. Interestingly, in the first defined variant [33] of this theory justifications were represented as trees [72] instead of graphs, whereas graph-like justifiactions were introduced later in [32].

6.8 Explaining unsatisfiability

A typical approach for several applications of ASP such as problem-solving, for instance, is the *generate-and-test* strategy. That is, the ASP specification defines a search space that is known to be a superset of the solutions, considers all of them possibly by generating them via choice rules, and finally those that are not considered a valid solution are disregarded. For the latter, integrity constraints are typically used to define what constitutes a valid solution. Take for instance the blocks world problem, for which we introduce an encoding for obtaining commonsense explanations in Section 5.3. This is a good example of the generate-and-test approach applied to problem-solving. Programs 5.3, 5.4 and 5.5 show different ways of generating the same search space of solutions, whereas in Program 5.2 several constraints test if the generated potential solution comply the blocks world rules or not (i.e. only blocks on top of others are moved, at the final step the goal state is reached, etc).

If generate-and-test is a common approach in ASP, it follows naturally that finding the reasons a particular potential solution is not a valid solution is of great importance for explainability in ASP. Moreover, sometimes these invalid solutions must be considered as a use case as in the case of configuration of products [83]. In this problem, an ASP program is used to consider the possibilities for configuring a product with respect to some user preferences. However, it may be the case where a user may introduce conflictive preferences, this is any configuration that complies with them is possible. In such a case, we would like to inform the user of the reasons why this happens, either based on the user's preferences or the product configuration constraints. In all cases, since this is a user-use case, it would be desirable to provide common sense explanations for this.

This means that rather than only asking *how-come* or *why not* questions about events that happen or not in the current solution it is also of great importance for explainability in ASP to be able to answer questions similar to *why does this instance does not have a solution?* or *why this <query> is not a solution?*. As we already introduced in Section 5.5, note that identifying them as different solutions does not mean that they cannot be implemented in a similar way. In fact, technically speaking, this problem can be stated as the problem of finding the reasons why a potential answer set is not a solution, or what we have to change in the program for one of these sets to be a valid solution. Thus, this relates to the fields of debugging ASP programs [16, 97] and repairing ASP programs [1, 105]. In general terms, in both cases, there is an expected output that a program is not producing and the problem is to identify which parts of the program are responsible for that. In the case of debugging, the modification of the identified bug is left open to the engineering user, whereas in repairing, the idea is further suggesting (typically the minimal set of) changes in the program so that the expected output is obtainable.

For now, most of the research has focused on producing technical explanations, directed either to knowledgeable ASP engineers or to the system itself, but little research has been done on its use for generating user-friendly explanations. In Section 5.5.3, we propose an implementation for obtaining how-to explanations with `xclingo` that is related to the topic and could be indeed considered a form of repairing. Further research on this topic exceeds the goals of this dissertation but it represents a line of future research of great interest and applicability. For instance, in Section 5.5 we discussed the possibility of using abstraction methods [89, 90] to identify the relevant parts of a program such that it is inconsistent, and to use that for generating explanations. The rest of this Section visits some approaches for obtaining the reasons why a program is inconsistent or unsatisfiable.

From `spock` to DWASP

`spock` [16, 56] is a system for identifying semantic errors in ASP specifications. It relies on the use of a meta-program or reification that represents the tested specification but extends it with rules that identify the different reasons why a set of atoms (i.e. a potential answer set) may not be an answer set. In particular, it includes predicates saying if a particular original rule is applicable or not, another for saying if an atom is unsupported or not and finally one to say if an atom is unfounded or not. It also includes choice rules generating the considered potential answer sets so that the models of the reified program are invalid solutions accompanied by extra predicates indicating the reasons why they are not consistent.

One main drawback of `spock` is that only deals with propositional programs and not with ASP programs including variables. This problem was addressed later by the tool `Ouroboros` [76] which extends `spock`'s approach. One of the differences introduced is that, instead of generating a space of invalid solutions, this tool requires the user to introduce the potential model in question. That is, the approach assumes there is a solution that the user expects that is not a solution of the tested ASP program, which closely relates to contrastive explanations. To this aim, this tool extends the `spock`'s reification, and introduces the expected model as a predicate, similarly to what `xclingo` do with the model to explain (see Section 4.3.1).

`Ouroboros` requirement of providing the whole model was a bit cumbersome for the user. For this reason, the approaches that followed switched to an interactive approach. This was de case of the work presented by [92]. In this method, the user is asked by the system only for the relevant atoms that the user expects to be in the solution, and the rest of the atoms are generated like in `spock`. This work introduces the fundamental concepts of *test cases* and *background theory*. Test cases are sets of atoms, in particular, it defines four sets: *positive cases* which are two sets of atoms that must be in (respectively out of) all answer sets; and *negative cases* which are sets of atoms that must be in (respectively out of) some answer set. The background theory is a subset (of rules) of the tested program. It relies on the same predicates that `spock` use to find unsatisfied rules and unsupported and unfounded atoms. The atoms belonging to those predicates are called *abnormality atoms*. The idea behind the method is to identify the sets of abnormality atoms that satisfy the test cases and the given background theory.

Later, another tool called DWASP [35, 50] was published. This is a debugging tool that leverages the solver WASP [2, 3] and applies some of the ideas introduced by the previously commented approaches, such as the use of a background theory or the creation of a reification of the program called *debugging program*. In this reification, an ad-hoc *debug atom* is introduced in the body of any rule that does not belong to the background theory, that defaults to the set of atoms in the program. This default atom identifies the rule they participate in and contains all the variables in the body of such a rule. The debugging program is then solved with WASP, which allows one to assume the truth of some atoms when computing the answer sets. Since debug atoms only appear in the body of rules, they are unsupported by default, meaning that any rule with any of these atoms in the body can be true unless its corresponding debug atom is assumed true. DWASP exploits this by computing unsatisfiable cores. Unsatisfiable cores are sets of atoms that, if true, prevent the program from having any answer set. Leveraging this notion, DWASP finds the subset of debugging atoms such that, if one of them is not assumed to hold, then an answer set of the program exists.

Perhaps is of interest to discuss the different meta-programming approaches. As we have seen, all the previously mentioned approaches use a meta-programming method in some way. This is true also for other explanability systems like `xclingo` (see Section 4.3) and `xASP` (see Section 6.4). All of these systems produce a meta-encoding or reification corresponding in some way to the program that is being debugged or explained. Although different, they all seem to represent the same notions of the supportiveness of an atom, the satisfiability of a rule, and so on. For instance, note the similarity between the DWASP's *debug atom* and `xclingo_xclingo_sup` predicate. In both cases, they are associated with a particular rule and they contain the variables of the body of such rule. The aim of the latter is to collect the values the rule was derived with, in order to show hints during debugging (in the case of DWASP) and for designing better natural language explanations in the case of `xclingo`. Moreover, it seems that even though most explainability approaches focus on technical explanations almost all of them are trying to emulate the underlying semantics of the language rather than explaining how the used solver works. It could be of interest to have, in addition to user-oriented and semantic technical explanation layers, a deepest solver technical layer that speaks in terms of the solver steps. Perhaps the focused interest of the research in semantic technical explanations is due to the fact that most ASP engineers are neither solver developers nor final users, but rather they fall in the middle.

Part II

Applications to Explainable Machine Learning

Chapter 7

A Tool for Explaining Decision Trees Applied to Liver Transplantation

7.1 Introduction

In Part I of this thesis, we have explained our approach to explaining models of logic programs and provided a tool that implements such an approach (in particular for ASP programs). There are a lot of questions that still wait to be answered in that context and, in recent years, the interest in the topic is growing in the field of Logic and Knowledge Representation. Thus, a very important part of the research effort included in this thesis is towards the development and application of `xclingo` for Logic and KR contexts. However, another important part of the work was more related to the application of `xclingo` for obtaining explanations in ML contexts. In particular, two different case studies were considered. The first of these two is the development of a Support Decision System for liver donor-recipient matching of liver transplants. In a critical context like this one, there is no discussion of why explanations are needed. Besides, it is particularly important that such explanations are reproducible and that provide accountability and transparency. The developed system provides estimations of the survival of potential donor-recipient pairs, as well as explains such estimation. The explanations are obtained using `xclingo` from an ASP representation of an ML classifier, in particular a Decision Tree Classifier (DT). More precisely, a tool called `Crystal-tree` was developed, acting as a client of the `xclingo`'s Python API, that provides natural language explanations of trained DT models. In this Chapter, we present such a tool and explain how it works, including the logic programming implementation using `xclingo`.

The rest of the Chapter is structured as follows. Section 7.2 introduces the problem of donor-recipient matching and briefly explains how the ML learning models were obtained. Section 7.3 explains the architecture of the developed system and how `cystal-tree` is used to obtain explanations, as well as shows an example of use. Finally, Section 7.4 shows how `Crystal-tree` is used and explains the particular logic programming implementation using `xclingo`.

7.2 Machine Learning Models for Utility Estimation in Liver Transplantation

One of the most critical and sensitive medical procedures is the management of the waiting list in a transplantation unit. In the case of liver transplantation, the usual approach for donor-recipient matching establishes a priority in the waiting list through some scoring system such as MELD [104], [63] (Model for End-Stage Liver Disease), SOFT [86] (Survival Outcomes Following Liver Transplantation), or BAR [37] (Balance of Risk). These scores are mostly focused on the seriousness of the patient's condition (for MELD, this is the only criterion), giving less relevance to the suitability of the matching with a given donor. This approach leads to undesirable side effects. On the one hand, new patients entering the list with higher (that is more urgent) scores get incoming grafts more quickly, but due to their worse health state, this often implies a shorter average expansion of the recipient's life span. On the other hand, the recipients that would have lasted longer, stay longer periods on the list, waiting for their health state to become "worse enough" to be selected to get a graft. In other words, while the urgency-based approach minimizes the deaths within the waiting list, the average life span of the recipients could be increased by more utility-oriented criteria. The *utility* of a transplant is usually measured as the time elapsed since transplantation until a graft failure or, ultimately, the patient's death, within some predefined temporal window. Thus, in utility-based approaches, the adequacy of the matched graft acquires a crucial role: a particular recipient may have a longer estimated life span with one graft than with another and, in the same way, one graft could grant a longer life span to one patient than to another. To put an example, this would allow us to avoid assigning an incoming graft to the most urgent patient when there is a high risk of failure for that particular matching when compared to other patients on the list. Of course, the problem then becomes how to estimate that risk.

Computers can be used to assist the specialists in deciding a suitable donor-patient matching by predicting the result of the transplantation using some Machine Learning (ML) method trained on the available data. In this way, the predicted utility (possibly among the most urgent recipients) could complete the current protocols, or could even be incorporated into them as one more factor in the scoring systems. Indeed, several examples of applications of ML to the transplantation domain have been published in recent years. While some of them, like [67], focus on predicting the short-term survival of the recipient, others focus more on the long-term like [61].

It seems pretty obvious that a protocol for donor-patient matching cannot eventually rely on a black-box system, regardless of the potential accuracy of its predictions. The system behavior has to be accountable for healthcare professionals, patients and relatives or otherwise, they would not trust it. Also, it must guarantee fairness and transparency, even from a legal point of view. Given that most ML algorithms, particularly those with the highest performance, behave as black boxes whose conclusions can be hardly explained or not explained at all, this problem is perhaps the most important challenge yet to be faced when applying these ML techniques. It may be even more important than dealing with clinical data, which contains tons of sensible data and is often stored in an unstructured way, on multiple platforms.

Thus, we opted for a hybrid approach to build our system where two ML models are trained. In our hybrid approach, one opaque (possibly superior in terms of performance) model provides statistically trustable predictions while a second transparent model supports it with a second (this time explainable) prediction.After testing out several ML models, we selected a Multilayer Perceptron (MLP) model for the former, while for the latter, we used a Decision Tree (DT) model. The reason behind using DT models is that they can be represented as a set of readable symbolic conditions or rules, meaning that one can write an ASP program and use `xclingo` to obtain natural language explanations from the tree. Besides, in the recent survey [29] of articles that apply AI to organ transplantation, more than half of the approaches include a DT-based learning algorithm. Thus, our approach could be exploited in already deployed systems that use DT models to enhance them with natural language explanations.

To the aim of training such models, an important amount of data was manually collected at the liver transplantation unit at the University Hospital Center[1] of A Coruña (CHUAC), Spain. This data was properly analyzed and preprocessed for the ML algorithms. The best-predicting input features of the dataset were found, via both univariate and multivariate statistical significance analysis. Three algorithms were tested: Decision Tree, Multilayer Perceptron and Random Forest, for which their hyper-parameters were optimized. After the analysis, we obtained the best models for each algorithm, whose comparison in performance is shown in Figure 7.1. From the results, it is easy to see how

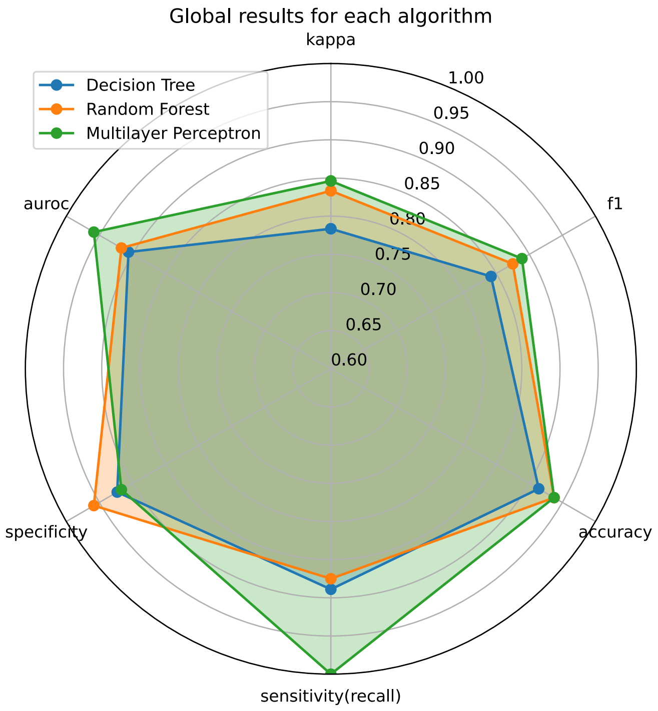


Figure 7.1: Best results for each algorithm. Several metrics are shown, including some typical commonly used in the medical field like sensitivity and specificity.

MLP outperformed the rest of the algorithms and thus was finally selected as the opaque model. On the other hand, the disparity between the DT results and the other algorithms is clear. However, this difference is mitigated by its capacity to produce explanations, something that is missing in both RF and MLP models. Besides, is important to underscore that the alignment of the explanations provided by the particular DT with the user requirements, is equally crucial as the performance of the models.

[1] *Complexo Hospitalario Universitario de A Coruña*, CHUAC.

Unfortunately, the explanations of our best DT model were not approved by our medical expert. According to the evaluation, even when there was an agreement between the system and the expert, the arguments exposed by the explanations were not clinically significant, resulting in a decrease in trust from the expert's standpoint. In pursuit of a valid DT model to obtain explanations from, the top 7 DT models found were detailedly analyzed by the expert. In a new methodology step that could be compared to what we have explained about *explanation design*, the expert selected one DT model among the 7 candidates, aiming to find a good tradeoff point between the performance of the model and the clinical significance of the explanations. Finally, the top 7 DT model was selected by the expert to be the *explainer model* of the system. The complete model can be read in Figure 7.2. The subset of features used by the tree and in the rest of the examples of this Chapter are described in Appendix D.

7.3 The Support Decision System

Figure 7.3 shows the main components of the final support decision system. From the particular donor-recipient pair, the corresponding input features are collected to create an input instance. As we can see in the figure, the system applies two different ML models, the DT (*Top7* from Figure 7.2) and the best MLP model, which respectively use their own subsets of input features. The *Supportive Explanation* is obtained from the tree by the use of the `Crystal-tree` Python package. Finally, both predictions as well as explanations are provided to the final user (i.e. the medical expert).

After processing a particular input instance, the system finally produces an output like what can be seen in 7.4. The report first shows the prediction of the DT model with a confidence level, in this case `dead` with a confidence of `100%`. Then, we get a natural language explanation extracted from the DT. The explanation is generated by `crystal_tree` by traversing the conditions in the DT path, summarising them and displaying text patterns to obtain a readable set of sentences. As an additional reminder, we also show the metric results we obtained for the DT model on the test set – this part is fixed, as it does not depend on the input instance. The MLP prediction is similarly shown below, but it does not include the textual explanation. In this case, the MLP agrees with the DT in a `dead` prediction with a slightly lower confidence of `93.37%`. Note that, in general, we may have cases in which the two models disagree, since these models have different predictive power. In such a case, reliance on confidence ratios, model-specific performance metrics and the provided explanation serves to help expert evaluation of the case on hand. In this way, we aim to sensibly manage the user's trust in the system for each scenario, rather than imposing reliance on an oracle-like prediction. For instance, the expert can see her initial intuition reinforced when the two models agree showing a high confidence and a clinically sensible supporting explanation, that is, a relevant combination of conditions that could have escaped to the expert's attention at first sight. Conversely, it could undermine the expert's intuition when predictions conflict with it, potentially leading the user to reassess her original stance. Moreover, when confidence is low or the explanations lack significance, users are led to be cautious and withhold trust in the system for the specific input pair.

In particular, the explanation shown in Figure 7.4 comes from the leaf of the DT that is highlighted in Figure 7.5 and so that its 3 arguments use features *don_sodium*, *rec_sodium* and *rec_hcc_afp_30* which respectively correspond with the features used in the nodes of that branch of the DT ordered from the root to the leaf. From the 3 arguments of the explanation, the second (The recipient is hepatitis C-positive) and the third (The recipient has HCC and her afp is higher than 30 ng/ml) are indeed

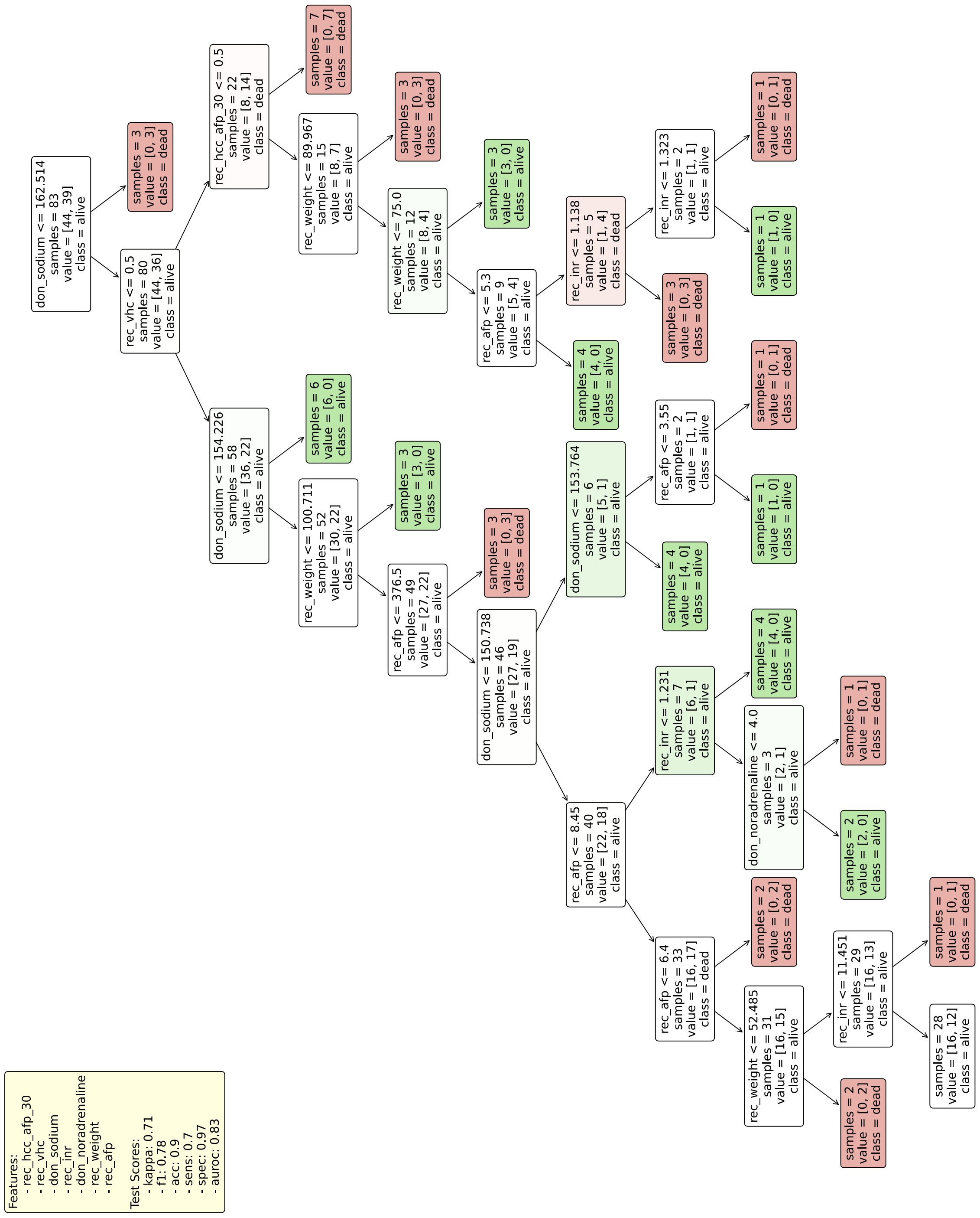


Figure 7.2: Decision tree *Top7* with kappa=0.71.

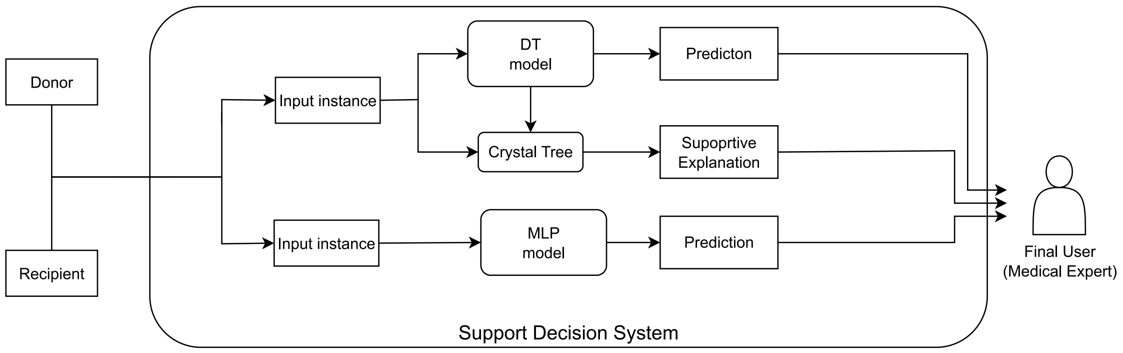


Figure 7.3: General view on the architecture of the final developed support decision system.

```
Decision Tree Predcition is: dead (confidence = 100%)
---------------------------------------------------------------------
Explanation:
    *
    |__Bad forecast: dead (confidence= 100%)
    |  |__The donor's sodium is lower than 162.51 mEq/L
    |  |__The recipient is hepatitis C-positive
    |  |__The recipient has HCC and it's afp is higher than 30 ng/ml

Model's performance:
    Kappa            0.71
    Sensitvity       0.70
    Specificity      0.97
    AUROC            0.83


Multilayer Perceptron (MLP) Prediction is: dead (confidence = 93.37%)
---------------------------------------------------------------------
Model's performance:
    Kappa            0.85
    Sensitivity      1.00
    Specificity      0.92
    AUROC            0.95
```

Figure 7.4: Example of output of the system for a particular donor-recipient pair.

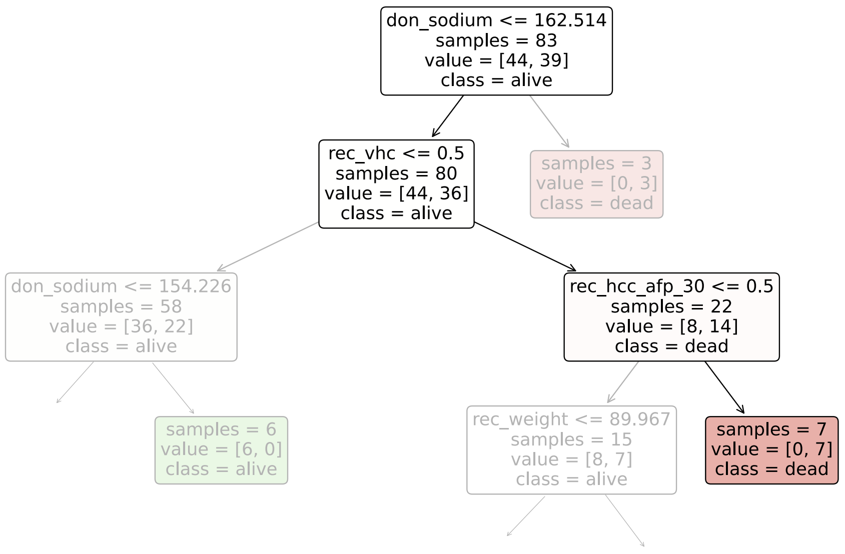


Figure 7.5: Branch of DT 7 (completely shown in Figure 7.2).

known reasons for a bad prognosis. On the other hand, for the first argument (The donor's sodium is lower than 162.51 mEq/L), this is not so clear. In general, we say a patient suffers from hyponatremia when sodium levels are under 135 mEq/L and from hypernatremia when are above 145 mEq/L. Saying that the donor's sodium is lower than 162.51 mEq/L includes both hyponatremia and hypernatremia ranges, as well as the normal values between 135 and 145 mEq/L. This makes the first argument not so clinically significant, but just a reflection of the conditions used by the DT to ultimately reach the conclusion. This is a clue of why the explanations obtained from any DT models, should not be understood as causal explanations, rather than an explanation of how a model achieved a certain prediction. In fact, having the sodium being above the threshold (note that this is a counterfactual question), the conclusion would have been the same, meaning that the condition was not causally necessary. Section 7.6 further develops on how to understand (or not to) the explanations obtained from a DT.

7.4 Crystal Tree: A Tool for Explaining Decision Trees

In this section, we describe our tool called `Crystal-tree` which produces text explanations from a given decision tree and a set of values for the input features. The explanations include the outcome of the decision tree together with those conditions satisfied by the input values that led to the conclusion. These explanations are provided in natural language, in a summarized way as shown, for instance, in Figure 7.7 used to explain a path from one of the discarded DT models, displayed in Figure 7.6.

The explanations must be read as a summarized representation of the path followed by the input sample (i.e. a recipient-donor pair) until a decision node is reached. The explanations produced by `Crystal-tree` are text-based and are organized in a two-level-tree structure. The first level shows the decision of the tree (for Figure 7.7 the recipient will survive more than 5 years) for the given input and the probability of the decision, according to the data observed by the DT during training. The second level summarizes the conditions met by the input sample (i.e. the recipient) to follow a certain path and

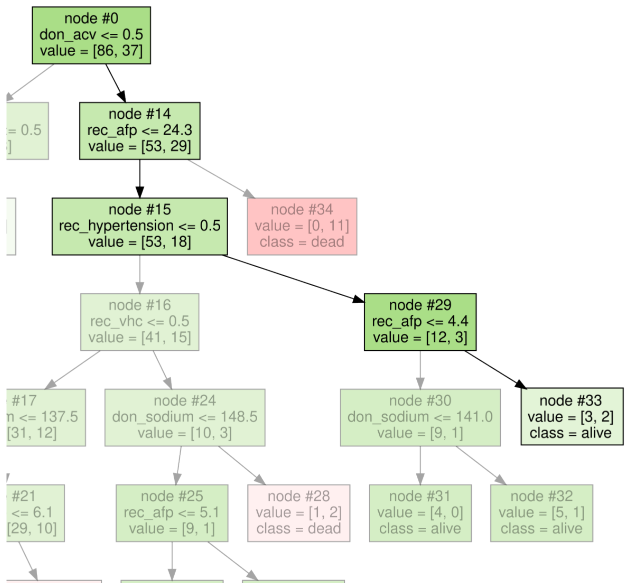


Figure 7.6: An example of a branch in one of the (disregarded) Decision trees.

```
*
|__Good forecast: alive, for recipient 126 (confidence: 60\%)
|  |__The donor's cause of death was a CVA
|  |__The recipient is hypertensive
|  |__The recipient's level of AFP is between 4.4 and 24.3
```

Figure 7.7: Example of explanation corresponding to the path in Figure 7.6.

```
*
|__Good forecast: alive, for recipient 126 (confidence: 60\%)
|  |__Donor death: stroke
|  |__The recipient is hypertensive
|  |__Tumor marker status: good
```

Figure 7.8: A version of the explanation in Figure 7.7 for non-specialists.

```
*
|__Buen pronóstico: vivo, para el receptor 126 (confianza: 60\%)
|  |__Muerte de donante: derrame cerebral
|  |__El receptor es hipertenso
|  |__Estado del marcador tumoral: bueno
```

Figure 7.9: Spanish version of the explanation in Figure 7.8.

to produce the final decision. The latter includes only one sentence per each feature used to reach the decision, so that: (1) features not used in the path of the tree are not included in the explanation; and (2) all the conditions over a repeatedly used feature within the path are summarized as the narrowest range of all the thresholds. As an example, note that the explanation shown in Figure 7.7 does not mention the medical condition of the recipient because it does not appear in the path (Figure 7.6) used to decide the outcome. Also, despite the path includes two conditions about the recipient's alpha-fetoprotein (AFP) (less or equal to 24.3 and more than 4.4) both of them were summarized in just one sentence. This is particularly helpful when dealing with deep trees that use a few variables. In those cases, we may end up having paths that include a lot of conditions over the same variable and gradually close the interval of values that led to classification. The summarization of such conditions makes the explanation easier to follow.

By default, the tool generates domain-independent explanations, including conditions that just refer to the variable name and its possible values, like for instance, `don_cva = 1`. However, the user can tune the text in the explanations using some sort of template: note how `don_cva = 1` is replaced by the text '`The donor's cause of death was a CVA`' in the explanation of Figure 7.7. This feature can be used for adapting the explanations to different kinds of situations, or users. With the same decision tree and input data, the outcome explanation can be adapted to different technical levels, to give more or less detail (some variables can be hidden, even under certain conditions) or to be expressed in different languages. As an illustration, Figure 7.8 provides an alternative explanation intended for a non-specialist user, such as the recipient herself, whereas Figure 7.9 displays the same explanation but generated using the Spanish text labels.

The behavior of `Crystal-tree` is controlled through a `Python` library, so other kinds of adaptations are potentially possible. For instance, considering again the translation example, the explanations could be dynamically translated into any language selected by the user. Figure 7.10 shows an example of usage of the `Python` library. From lines 1 to 9, the code loads the data and trains a Decision Tree classifier

```
1  import sklearn
2  from Crystal-tree import CrystalTree
3
4  # Loads a dataset
5  X, y = sklearn.datasets.load_iris(return_X_y=True, as_frame=True)
6
7  # Trains a decision tree
8  clf = sklearn.tree.DecisionTreeClassifier()
9  clf.fit(X,y)
10
11 # Translates the classifier into an explainable logic program
12 crys_tree = CrystalTree(clf)
13
14 # Creates the labels used by the tree
15 # if skipped, default labels will be used
16 setup_traces(crys_tree)
17
18 # Print explanations for input X
19 crys_tree.explain(X)
```

Figure 7.10: Example of use of the `Crystal-tree` library.

(which is saved in the variable `clf`) on the whole data set. Line 12 creates a `CrystalTree` object from the trained decision tree. Line 19 generates an explanation for each input sample in the vector `X`. As explained before, by default, `Crystal-tree` will use generic text for generating the explanations. To adapt the text to the domain, additional `Python` code is needed. The function `setup_traces` in line 16, adapts the explanations to get the same result as shown in Figure 7.7. Figure 7.11 shows part of the code within the function. To personalize the explanations, `Trace` objects are added into the `Crystal-Tree` object. This will introduce new text in the explanations, which will overwrite the default. These `Trace` objects can be associated either to a certain target class, (when the `target_class` argument is provided as in lines 4 and 7), or to a certain feature (when the second argument is the name or the ordinal position of a feature), for which a set of conditions can be further set. If the feature is used in the corresponding path and the provided conditions are satisfied by the input sample, then the explanation will include the trace text. The specified text pattern can also include four special placeholders, `%_class`, `%_instance`, `%_prob`, `%_t`, `%_min` or `%_max` that will be replaced by the names of the class, the identifier of the input sample, the probability of the prediction or the value of the threshold used, respectively. More detailed instructions on this topic (including examples) together with the source code and installation instructions are available in the `Crystal-tree` public GitHub repository[2].

[2]All files are publicly available in `https://github.com/bramucas/Crystal-tree`

```python
from Crystal-tree import Trace, Condition

def setup_labels(crystal_tree_object):
    crystal_tree_object.add_trace(
        Trace("Good forecast: %_class, for recipient %_instance (confidence of %_prob)",
            None, target_class=0)
    )
    crystal_tree_object.add_trace(
        Trace("Bad forecast: %_class, for recipient %_instance (confidence of %_prob)",
            None, target_class=1)
    )

    crystal_tree_object.add_trace(
        Trace("The donor's cause of death was not a stroke",
            "don_cva", predicate="le", threshold=0.5)
    )
    crystal_tree_object.add_trace(
        Trace("The donor's cause of death was stroke",
            "don_cva", predicate="gt", threshold=0.5)
    )
    crystal_tree_object.add_trace(
        Trace("The recipent's weight is between %_min kg and %_max kg",
            "rec_weight", predicate="between", thresholds=(60, 80))
    )
    crystal_tree_object.add_trace(
        Trace("High level of liver tumor marker in the recipient (afp above 30 ng/dL)",
            "rec_afp", conditions=[Condition("rec_afp", ">30.0")])
    )
    ( . . . )
```

Figure 7.11: Adding labels to a `Crystal-tree` object.

7.5 Implementation using Xclingo

At an implementation level, `crystal-tree` leverages the fact that a DT classifier can be formalized as a formula in Disjunctive Normal Form (DNF). Essentially, this formalization involves identifying all the leaf nodes that lead to the prediction of a particular class and collecting all the conditions $c_1 \dots c_n$ considered in the path that comes from the root of the tree. Each path is then formalized as the conjunction of all the conditions $c1 \wedge c_2 \wedge \dots \wedge c_n$. The disjunction of all the formalized paths is DNF logic formula that captures the behavior of the classifier for the given class. In the case of binary classification DT models, the negation of such a formula captures the behavior of the opposite class. For multiclass classification problems, it becomes necessary to formalize each class's particular formula to completely capture the classification function, in a *one against all* way.

Figure 7.12 shows how the Python package processes the data to obtain the explanations. `crystal-`

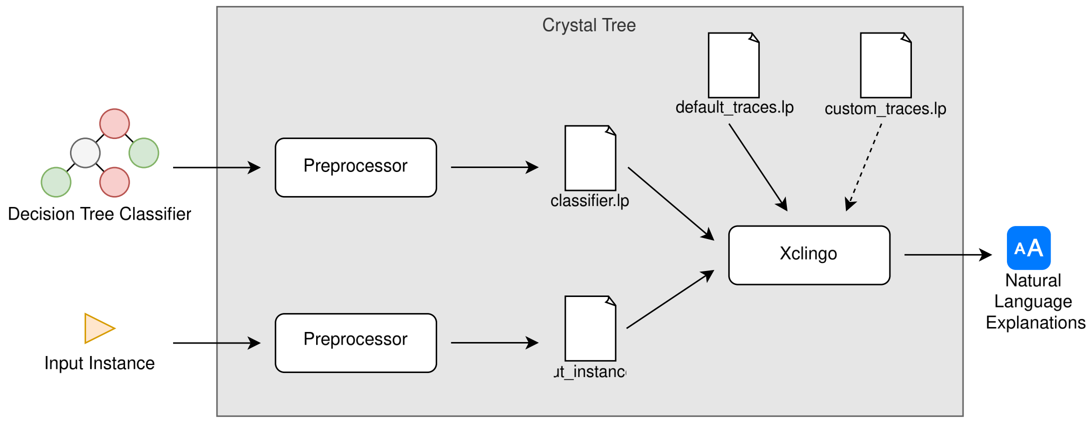


Figure 7.12: The data flow of the `crystal-tree` Python package to obtain explanations from a DT model and an input instance.

`tree` takes a DT model, computes such a formula in the form of a `xclingo`-annotated ASP program, and then uses `xclingo` to obtain natural language explanations for given input instances, which involves the encoding of the input instance as a set of ASP facts as well. Both pieces are put together with some `xclingo` annotations (in the Figure, *default_traces.lp* or *custom_traces.lp*, depending on the user's configuration) and then are explained by a simple call to the `xclingo` Python API.

Implementing an *ad hoc* procedural algorithm from scratch for the same purpose would not be too difficult. The main advantages of the logic programming implementation, however, are its flexibility and declarativeness. This has multiple potential advantages, like enabling variations of the classifier behavior by simple changes in the logical rules, studying the classifier results under uncertainty (i.e. when some variables are unknown) or allowing formal comparisons (like the analysis of equivalence) to other DTs, when represented as logic programs. Indeed, `crystal-tree` also provides the option of just obtaining the encoding for the user to modify any part of it or use it for other purposes different from obtaining explanations.

Let us illustrate how the ASP representation for the DT models works. Take, for instance, a particular donor-recipient pair with the following particular values: `don_sodium = 157.054`, `rec_vhc = 1` and `rec_hcc_afp_30 = 1`. Each value for each variable is converted into a fact using `value(I, F, V)` predicate, where `I` is an identifier for the input instance (one can provide several instances at once and obtain several explanations), `F` is an identifier for the particular feature (this identifier is defined at the moment of generating the translation) and `V` represents the value of the instance for that feature. The particular translation of such an input instance is shown in Figure 7.13. First note that, for brevity,

```
value(1, 2, 157054). % don_sodium      157.054
value(1, 1, 1000).   % rec_vhc           1.0
value(1, 0, 1000).   % rec_hcc_afp_30    1.0
```

Figure 7.13: Partial translation of an input instance for DT *Top 7*, depicted in Figure 7.2.

we are only considering 3 of the seven features that our DT model uses. A complete translation of an input instance would use all of them. As it is the first input instance that we are asking to explain, it receives the identifier `1`. Another important detail is the treatment for decimal values. As we cannot represent them directly in the ASP specification, in the moment of translation all numerical values are scaled (i.e. multiplied by some previously computed factor) to preserve the precision used in the numerical representation of the tree. This transformation is reverted once the textual explanations are built.

For preprocessing the DT model an ad hoc Python code *inorderly* traverses the tree keeping track of the conditions in each path. Whenever it reaches a leaf node, it generates a rule that represents that path such that if the values of an instance meet the conditions (i.e. the particular donor-recipient pair would *traverse* that path in the tree), the rule is fired. For instance, for the path in Figure 7.6, the preprocesser generates the rule in Figure 7.14. The head of the rule uses `class(C, I, P)` predicate,

```
class(1,I, 100) :- gt(I,0,500), gt(I,1,500), le(I,2,162514).
```

Figure 7.14: Translation for the path in Figure 7.6, from DT *Top 7* (Figure 7.2).

where `C` is the identifier for the class that the represented path is predicting, `I` is the input instance that is being classified (i.e. the body of the rule is true for instance `I`) and `P` is the probability associated with the decision of the leaf node. The body of the rule is the conjunction of all the conditions in the particular path. To that aim, three predicates are used: `gt(I, F, T)` that means that the value of instance `I` for feature `F` is greater than threshold `T`, `le(I, F, T)` which represents the complementary *lower or equal to* condition, and finally `between(I, F, Min, Max)` which imposes the feature `F` to be within the range `(Min, Max]`. The definitions for these predicates are shown in Figure 7.15. These three rules are static

```
gt(I, F, T) :- value(I, F, V), V>T, thres(T).
le(I, F, T) :- value(I, F, V), V<=T, thres(T).
between(I, F, Min, Max) :- value(I, F, V), V>Min, thres(Min), V<=Max, thres(Max).
```

Figure 7.15: Static base code containing the definitions of predicates `gt/3`, `le/3` and `between/4`.

code that is included in any translation for any DT model. Note the use of predicate `thres(T)`. It is used to provide ground to the rules and it represents the whole set of different thresholds used in the nodes of the tree. For the given DT model, a set of `thres/1` facts is generated. For instance, for DT *Top 7* from Figure 7.2 the set of facts in Figure 7.16 is generated. As for the input values and the conditions in each

```
thres(500).    thres(162514). thres(154266). thres(100711).
thres(150738). thres(11451).  thres(52485).  thres(6400).
thres(8450).   thres(4000).   thres(1231).   thres(153764).
thres(3550).   thres(100711). thres(5300).   thres(1323).
thres(1138)    thres(376500).
```

Figure 7.16: The set of `thres/1` facts generated for the DT model *Top 7* depicted in Figure 7.2.

path, the values are scaled to preserve the decimal precision. As was explained before, the representation used by `Crystal-tree` summarizes repeated conditions over the same variable. For instance, for a path having the conditions `don_sodium <= 162.514` and `don_sodium <= 154.226`, our representation would only include the latter condition, which is stronger. This summarization may involve imposing a closed range over a variable, in which case the `between/3` predicate is used for that variable.

For an example of a full translation see Proram 7.1, which contains the full translation of DT model *Top 7* from Figure 7.2. The code shown constitutes an example of a *classifier.lp* program, which is one of the main components used in Figure 7.12. Note that, the last line, contains a `show_trace` annotation for querying explanations for any instance `I`. It is a standalone ASP representation of the classifier and can be used for any purpose beyond making (or explaining) predictions. Indeed, the `Crystal-tree` tool can provide this translation to the user to be used as a base to implement other systems that reason about the classifier.

With a mere call to `xclingo` Python API the *classifier.lp* together with an encoded input instance would be enough to provide explanations by using the `xclingo`'s *auto-tracing* feature. However, the obtained explanations would be in terms of the atoms representing the conditions and so would fain in providing accountability. To solve this, a third piece of code containing some annotations is added to the generated program before calling `xclingo`. If the user does not define custom text traces, a default set of trace annotations is added. This default set is shown in Figure 7.2. The additional `feature_names(F, FName)` predicate is generated whenever the user provides `Crystal-tree` with the names of the input variables of the model. The default text traces change depending if there is a feature name available to be used or not.

If, on the other hand, the user defines any custom trace, the default will not be appended to the final translation and the traces following the user's definition will be generated instead. For instance, for the set of custom traces defined by a user in Figure 7.11, Figure 7.17 shows their corresponding translation into `xclingo` annotations. Depending on the user's configuration either `prediction`, `le`, `gt`, `between` or even `value` atoms can be traced with custom text. This provides good flexibility to customize the explanations. Also, when no feature names are provided, the ordinal identifier of the rule is used instead in the generated annotations.

7.6 Discussing Decision Tree Explanations

Even though the good performance achieved by the trained ML models, the critical point of this support decision system is the trust that the expert (final user) can place on it. In that sense, the first hurdle

```
%%% thresholds
thres(500).    thres(162514). thres(154266). thres(100711).
thres(150738). thres(11451).  thres(52485).  thres(6400).
thres(8450).   thres(4000).   thres(1231).   thres(153764).
thres(3550).   thres(100711). thres(5300).   thres(1323).
thres(1138).    thres(376500).
%%% base
gt(I, F, T) :- value(I, F, V), V>T, thres(T).
le(I, F, T) :- value(I, F, V), V<=T, thres(T).
between(I, F, Min, Max) :- value(I, F, V), V>Min, thres(Min), V<=Max, thres(Max).
%%% paths
class(1,I, 100) :-
    le(I,1,500), le(I,2,150738), le(I,5,52485), le(I,6,6400).
class(0,I, 57)  :-
    le(I,1,500), le(I,2,150738), le(I,3,11451), between(I,5,52485,100711), le(I,6,6400).
class(1,I, 100) :-
    le(I,1,500), le(I,2,150738), gt(I,3,11451), between(I,5,52485,100711), le(I,6,6400).
class(1,I, 100) :-
    le(I,1,500), le(I,2,150738), le(I,5,100711), between(I,6,6400,8450).
class(0,I, 100) :-
    le(I,1,500), le(I,2,150738), le(I,3,1231), le(I,4,4000), le(I,5,100711), between(I,6,8450,376500).
class(1,I, 100) :-
    le(I,1,500), le(I,2,150738), le(I,3,1231), gt(I,4,4000), le(I,5,100711), between(I,6,8450,376500).
class(0,I, 100) :-
    le(I,1,500), le(I,2,150738), gt(I,3,1231), le(I,5,100711), between(I,6,8450,376500).
class(0,I, 100) :-
    le(I,1,500), between(I,2,150738,153764), le(I,5,100711), le(I,6,376500).
class(0,I, 100) :-
    le(I,1,500), between(I,2,153764,154266), le(I,5,100711), le(I,6,3550).
class(1,I, 100) :-
    le(I,1,500), between(I,2,153764,154266), le(I,5,100711), between(I,6,3550,376500).
class(1,I, 100) :-
    le(I,1,500), le(I,2,154266), le(I,5,100711), gt(I,6,376500).
class(0,I, 100) :-
    le(I,1,500), le(I,2,154266), gt(I,5,100711).
class(0,I, 100) :-
    le(I,1,500), between(I,2,154266,162514).
class(0,I, 100) :-
    le(I,0,500), gt(I,1,500), le(I,2,162514), le(I,5,75000), le(I,6,5300).
class(1,I, 100) :-
    le(I,0,500), gt(I,1,500), le(I,2,162514), le(I,3,1138), le(I,5,75000), gt(I,6,5300).
class(0,I, 100) :-
    le(I,0,500), gt(I,1,500), le(I,2,162514), between(I,3,1138,1323), le(I,5,75000), gt(I,6,5300).
class(1,I, 100) :-
    le(I,0,500), gt(I,1,500), le(I,2,162514), gt(I,3,1323), le(I,5,75000), gt(I,6,5300).
class(0,I, 100) :-
    le(I,0,500), gt(I,1,500), le(I,2,162514), between(I,5,75000,100711).
class(1,I, 100) :-
    le(I,0,500), gt(I,1,500), le(I,2,162514), gt(I,5,100711).
class(1,I, 100) :-
    gt(I,0,500), gt(I,1,500), le(I,2,162514).
class(1,I, 100) :-
    gt(I,2,162514).

prediction(I) :- class(C,I,P).
%!show_trace {prediction(I)}.
```

Program 7.1: An example of a *classifier.lp* (shown in Figure 7.12). In particular, this ASP code represents the complete translation of the DT model *Top 7*, depicted in Figure 7.2.

```
feature_names :- feature_name(F,FName).

%!trace {gt(I,F,T), "% > %_t", FName, T} :- feature_name(F,FName).
%!trace {le(I,F,T), "% <= %_t", FName, T} :- feature_name(F,FName).
%!trace {between(I,F,Min,Max), "% in (%_t,%_t]", FName, Min, Max} :- feature_name(F,FName).

%!trace {gt(I,F,T), "feature % > %_t", F, T} :- not feature_names.
%!trace {le(I,F,T), "feature % <= %_t", F, T} :- not feature_names.
%!trace {between(I,F,Min,Max), "feature % in (%_t,%_t]", F, Min, Max} :- not feature_names.

%!trace {prediction(I), "Predicted class % for instance % (% probability)", C, I, P} :- class(C, I, P).
```

Program 7.2: *default_traces.lp* program containing the set of default traces used to generate natural language explanations for the DT models.

```
%!trace {prediction(I), "Good forecast: %, for recipient % (confidence %_prob)",C,I,P} :- class(C,I,P),C=0.
%!trace {prediction(I), "Bad forecast: %, for recipient % (confidence %_prob)",C,I,P}  :- class(C,I,P),C=1.
%!trace {le(I, F, T), "The donor's cause of death was not a stroke"} :- feature_name(F, "don_cva"), T=5000.
%!trace {gt(I, F, T), "The donor's cause of death was stroke"} :- feature_name(F, "don_cva"), T=5000.
%!trace {between(I, F, Min, Max), "The recipent's weight is between % kg and % kg", Min, Max} :- feature_name(F, "rec_weight
      "), Min=60000, Max=80000.
%!trace {value(I, F, V), "High level of liver tumor marker in the recipient (afp above 30 ng/dL)"} :- feature_name(F, "
      rec_afp"), V>30000.
```

Figure 7.17: Translation of the custom traces defined by a user in Figure 7.11.

we dealt with was that the explanations obtained by our best DT model were not convincing to the expert from a medical point of view. This forced us to evaluate other DT models (those immediately worse in terms of performance) in search of clinically meaningful explanations. That led us to conclude that it is a mistake to understand the explanations extracted from the paths of a DT model as causal explanations. On the contrary, the conditions from the decision nodes actually reflect statistical information that was used during the learning process. The DT learning algorithm proceeds recursively, introducing a new condition that *splits* the current data set into two new subsets in a way that the entropy of the result is minimized (that is, the information gain is maximized). The larger the difference between the ratios for each class in a subset, the lower the entropy is. Informally speaking, the upper a condition node is in the tree, the more it helps to "*clarify the picture*" with respect to a statistical partition of the data set. In this way, this arrangement helps in minimizing the average number of questions (depth of the path) we make to obtain a prediction. However, it also produces curious effects when reading a particular path as an explanation for a prediction. A first counterintuitive effect is that the same variable may be checked several times (even more than twice) along a path depending on different threshold values Fortunately, this redundancy can be easily removed from the explanations, as our tool `Crystal-tree` does.

A second, and more important, counterintuitive effect that appears when explaining a prediction by simply reading a path is that some of the conditions we check along the path can be *causally opposed* to the prediction eventually obtained. To see an example, suppose we had a transplant case with the following data:

```
rec_hcv = 1
```

```
don_cva = 0
rec_afp = 27
rec_provenance = 0
```

We will use one of the DT models that our expert discarded during the selection process. If we follow the DT conditions using these data, we get the path in Figure 7.18 reaching node `#12`, whereas its corresponding explanation is displayed in Figure 7.19. While the explanation predicts a bad fore-

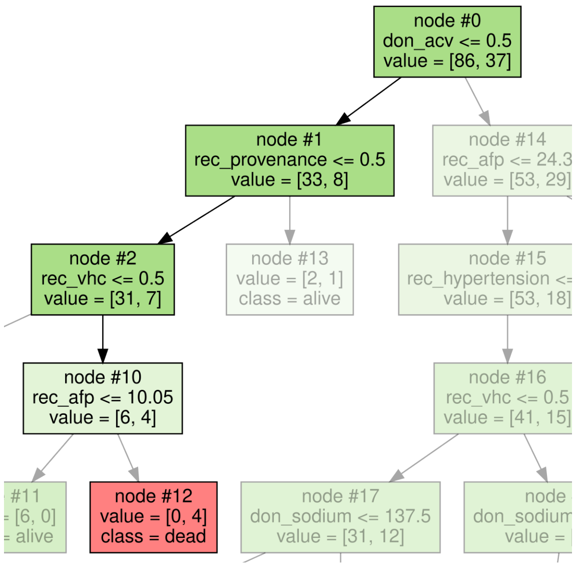


Figure 7.18: Another path in the decision tree, explained by Listing 7.19.

cast, the fact that the donor did not die by a CVA (`don_cva = 0`) and that the recipient is at Home (`rec_provenance = 0`) are known to particularly contribute to the good prognosis. These conditions must correspond to the nodes followed by the tree to make a prediction but *cannot be read as necessary causes* strictly required to achieve that prediction. In reality, what they do is restrict the subset of the data where the remaining conditions (which might be causally related) explain the prediction. For instance, this explanation could be understood as *Hepatitis C-positive recipient explains the poor prognosis*

```
*
|__Bad forecast: dead, for recipient 114 (confidence: 100\%)
|  |__The donor's cause of death was not a CVA
|  |__The recipient is at Home
|  |__The recipient is hepatitis C-positive
|  |__AFP is greater than 10.05 ng/ml
```

Figure 7.19: Explanation corresponding to the path in Figure 7.18.

among donor-recipient pairs in which the donor did not have a stroke and the recipient was not admitted to the hospital. Much of the information about these causal relationships is already known by the experts. This is, indeed, what allows them to discriminate between the DT models that are causally significant and those that are not. This *Background Causal Knowledge* information could be introduced in the form of text *Traces*. The causally opposed conditions of a path of the tree could be written in the explanations preceded by some predefined text like *"Despite of... ´´*, to give the hint that is not part of a causal relation discovered by the algorithm, but just statistical information it is leveraging form.

But perhaps a harder problem we may find when extracting explanations from the paths of a DT model is the existence of irrelevant conditions. Take for instance the selected DT depicted in Figure 7.2, and the donor-recipient pair with the following values:

```
don_sodium     = 152.500
rec_vhc        = 1
rec_hcc_afp_30 = 0
rec_weight     = 92
```

If one follows the conditions from the tree, this pair ends up reaching a *dead* leaf node after evaluating 4 condition nodes. Thus, a path-based explanation would consist of these four conditions displayed as equally valuable and necessary causes for the prediction, this is that prediction could not be achievable without meeting the conditions. Now see, however, that had the value for `rec_hcc_afp_30` be 1, our input instance would be as well classified as *dead*. Moreover, the same happens if increasing the `don_sodium` value to be above the first 162.514 threshold. What is more, in the three cases the associated confidence in the prediction is 100%. In other words, with the rest of the input data unchanged, *the prediction will still be* `dead` *regardless of these two variables.* To sum up, these two conditions in the explanation just reflect how the DT has organized the conditions to be checked, but cannot be read as necessary causes to explain the prediction.

An interesting possible future line of research is to design an algorithm that collects minimal sets of necessary conditions for each possible prediction made by the tree. Since we have already translated the DT model into a logic program, this is, into a logic formula in *Disjunctive Normal Form*, one option would be to minimize such a formula. To this aim, known approaches like Quine-McCluskey (QM) could be used to simplify this expression. The result of such a process, although would behave as the original DT model in any case, would not be a tree structure anymore but a set of rules whose bodies represent joint, necessary causes for the predictions. However, the QM method only works with Boolean expressions and, unfortunately, a lot of the conditions in a decision tree are ranges over numerical features. To apply such an approach, mapping the tree to a fully Boolean or adapting the QM algorithm to deal with ranges of numerical features would be needed. Another (similar in some sense) option to deal with unnecessary causes in DT path-based explanations is the one used in [30, 31]. From a given ML model, this approach computes *Symbolic Decision Graphs* from where obtaining explanations that avoid the use of any non-necessary input feature. These graphs can be represented as an ASP program to obtain natural language explanations via `xclingo`. Additionally, in this work, they also show how to obtain different kinds of formulas to perform counterfactual reasoning (i.e. finding out what changes we would have to make in the input to flip the actual decision of the classifier) or even find biases over specific input features.t

In contrast to the previously mentioned approach, a graph is obtained for each particular pair of models and input instance, whereas in the DT minimization approach, we obtain a fresh set of formulas representing the necessary conditions for classifying an input instance. Although our first attempts on this research line suggest that the computational cost of minimizing a DT model is much higher, this product is valuable and can be provided to the experts as a result.

Chapter 8

Explanations for ML Applied to 3D Printing of Medicines

8.1 Introduction

This chapter continues the research line obtaining common sense explanations for ML classifiers. In the previous chapter, we started with symbolic ML algorithms as DT. The problem then was to produce common sense explanations from such models that embrace the properties defined in Defintion 1 from Chapter 1. In this chapter, we face the task of obtaining explanations for non-symbolic classifiers.

Recall that one of the requirements for a common sense explanation is that it has to be causal. In the case of symbolic approaches, as they are naturally transparent (i.e. a technical user can just *read* what the classifier is doing), one can try to validate the causal knowledge that the model can be capturing from the correlations learned. For instance, in the previous chapter, we show how some of our DTs were directly disregarded by our expert because the rules learned by the tree were, in the expert's words, *not clinically significant*. From all the models, the expert chose one as the more clinically significant, that is, more aligned with the causal knowledge of the expert.

As you see, even though we cannot say DT models are causal (see Section 7.6 for a deeper discussion on this topic), we can at least check if their learned correlation knowledge is aligned with the user's intuitions. In the case of non-symbolic ML algorithms, that act as black boxes, this validation step can sometimes be too hard. Unfortunately, these systems are indeed those that were the most extended in recent years, and that are already affecting the lives of people on a daily basis (think on loan allocation or medical diagnosis, but also recommendations of all kinds such as in politics, culture or leisure).

In this Chapter, we study the application of a method for obtaining symbolic explanations for any kind of ML classifier. We provide an ASP implementation of such a method called `aspBEEF` and we test it in the real-world application of enhancing the development of 3D-printed medicines. In the next sections, we first introduce the study case of 3D-printed medicines research (Section 8.2) and after that we present the tool aspBEEF and the method it is inspired in (Section 8.3).

8.2 Accelerating 3D-Printed Medicine Research with ML

Since the approval of the first 3D-printed medicine, Spritam, 3D-Printing (3DP) has been touted as the next disruptor of the pharmaceutical manufacturing industry [98]. The gold end goal of this field of research is the so-called precision medicines, allowing every individual to be able to receive the right dose at the right time. Different (3DP) technologies have been tested for this purpose among them FDM is the most actively explored 3DP technology in pharmaceutics.

FDM is a thermal material extrusion technology whose popularity is mainly attributed to its affordability, versatility and compactibility [59]. Figure 8.1 outlines the typical process of FDM printing.

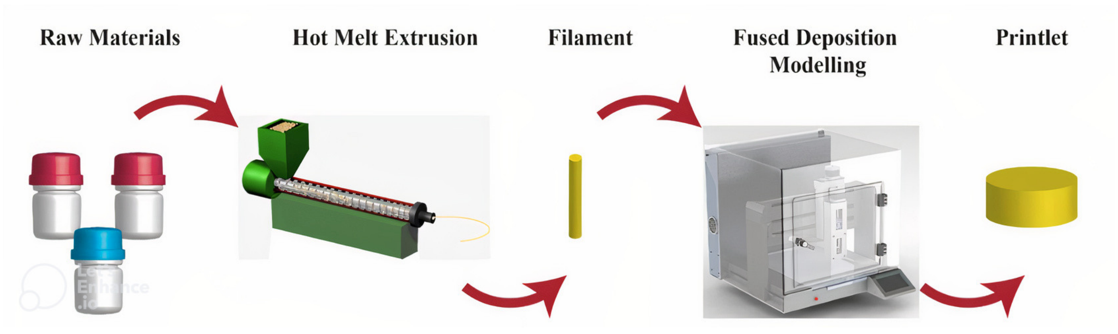


Figure 8.1: Typical process of FDM printing applied to 3DP of medicines. Medicines produced by this process are called *printlets* instead of tablets.

First, raw pharmaceutical material is processed through hot melt extrusion (HME) to obtain long strands of filaments which are subsequently fed into an FDM 3D printer. The printer melts the filament and it is deposited layer-by-layer onto a build plate to create a 3D object. The size and shape of the object can be easily modified by software.

Yet, developments of new medicines in pharmaceutical FDM 3DP have been hampered by the empirical process of formulation development. Numerous parameters within this two-step process can influence the performance of the final product. These include, but are not limited to, pre-HME variables (e.g. proportion of starting materials, object design), HME variables (e.g. extrusion temperature and speed, among others), and FDM 3DP variables (e.g. printing speed, printing temperature, platform temperature). Consequently, to produce the desired product, researchers must undergo a long process of trial and error, slowly adjusting each parameter one at a time and evaluating the performance of each prototype. Not only is this time-consuming and inefficient, but it also necessitates large amounts of material waste and monetary costs. Consequently, this results in a system unsuitable for on-demand applications. Therefore, it would be desirable to have a means of predicting the optimal parameters that will produce the 3D-printed object with the desired performance. Given a sufficient amount of real experiments ML algorithms may hold the key to optimizing this process.

In brief words, the goal is to train ML models able to predict the correct parameters that are needed to obtain a desired 3D-printed medicine, accelerating the trial and error process. Figure 8.2 outlines the two workflows that are the end goal of this application.

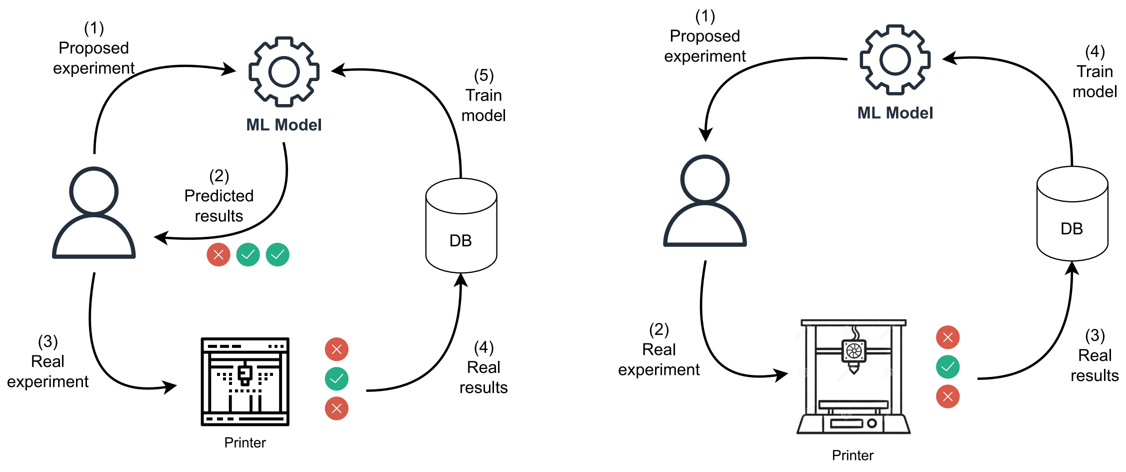


Figure 8.2: Experiments acceleration workflow (at the left side) and experiment recommendation (at the right side).

First, we propose the *Experiments acceleration workflow*, depicted on the left side of the figure. In it, the user (the laboratory operator) introduces the suggested experiment, that is the input parameters for the FDM process so that a successful experiment is expected. The system answers back with the predicted results for the experiment that could be successful or not. The operator continues to design new experiments until the system predicts that the experiment will succeed. Then the experiment is performed and the result, either good or not, is fed into the database so new models are trained. The hope is that the system helps in skipping failing experiments and even if some predictions are wrong and the experiment fails, that is added to the database of experiments as valuable evidence for training better models. Conversely, the second workflow, namely *Experiment recommendation workflow* requires the system to directly suggest the FDM parameters that will lead to a successful experiment. The operator performs the experiments and introduces the results into the system so it can learn more.

A third related goal is to obtain explanations for the system. To this aim, although several methods are yet to be tested, we have opted for obtaining a symbolic *Explainer*. Given a prediction of the classifiers, the symbolic explainer can produce an explanation in terms of numerical ranges over the input features. We later transform this output into a natural language message for the user.

Moreover, the explainer itself is a symbolic object (thus representable under a KR paradigm as ASP) that captures the behavior of the original ML model. In a sense, is a symbolic approximation of the ML model function that allows us to perform some reasoning in terms of how the model predicts a classification from a particular input. Recall now the recommendation workflow from Figure 8.2. In a sense, flow implies flipping the traditional way in which we leverage ML and, instead of predicting the output from the input parameters, we rather perform the counterfactual query *which input produces the desired output?* Having an approximated symbolic representation of the ML model allows us to perform this kind of task. This last approach is still untested and represents part of the future work of this project.

8.2.1 Obtaining the FDM-Predicting ML Models

Of course, to obtain the explainer object we need first to have trained and tested ML models with a good performance in predicting the experiment results. The experiments associated with the derivation of these models involved a very significant workload, which is beyond the scope of this dissertation. Even so, that process and its associated publications were produced also under this doctoral thesis, which is listed in Section 1.3. Still, work on explanations in this context is at an early stage.

For a detailed description of the data used, and the ML pipeline used to get the models, we refer to the published articles [24, 41, 77]. In the following, we give a very brief idea of the work done to obtain the models and the different related experiments.

3D-Printing Data

Unlike in the case study of Chapter 7, the data were collected and provided by the FabrX [1] [2] company although the design and development (and maintenance) of a relational database for this data was performed by the author of this dissertation. The dataset started comprising 614 experiments evaluated by expert HME and FDM operators from University College London – School of Pharmacy and the company Fabrx, using 143 excipients and drugs. These data grew with the subsequent publications, by collecting up to 980 experiments from almost a decade of 3DP of medicines literature, to reach the number of 1594 experiments and 254 different excipients and drugs. These data were analyzed by performing multiple tasks of visualization.

The evaluated input features can be classified into three groups: material concentration (that is, the relative amount of each material participating as part of the mixture before extrusion, see Figure 8.1), FDM input parameters and Dissolution parameters. From this source dataset, 5 variations were designed and processes for exporting the original dataset to the 5 formats were developed. Each variation encodes the material concentration in a different way, for instance, one groups the amounts by the type of molecule the material falls into; another for instance, such as the Physical Properties Dataset, expresses the mixture of materials as the weighted average of the individual physical properties of each participating material (their molecular weight, or their melting point, among others). Figure 8.3 depicts the alternative datasets and briefly outlines how they are obtained from the source.

ML Pipeline

Starting from these datasets, 4 ML algorithms were tested in the task of predicting the different 3DP parameters of interest, namely: (1) HM extrusion temperature; (2) HM extrusion filament aspect; (3) FDM printing temperature; (4) FDM *printability* (i.e. says if the obtained printlet is successful); and (5), dissolution profile. (1) and (3) are numerical variables, whereas (2) and (4) are categorical variables representing the quality of the extruded filament (and printed printlet, respectively) and (5) is the approximation of a function representing the dissolution of the printlet inside the patient body over time. The ML training pipeline was incrementally enhanced with the subsequent publications. At the moment, the pipeline consists of several steps including, among others, normalization and standardization of numerical variables, and exhaustive feature selection via grid search. Figure 8.4 shows

[1]https://www.fabrx.co.uk
[2]https://fabrx-ai.com

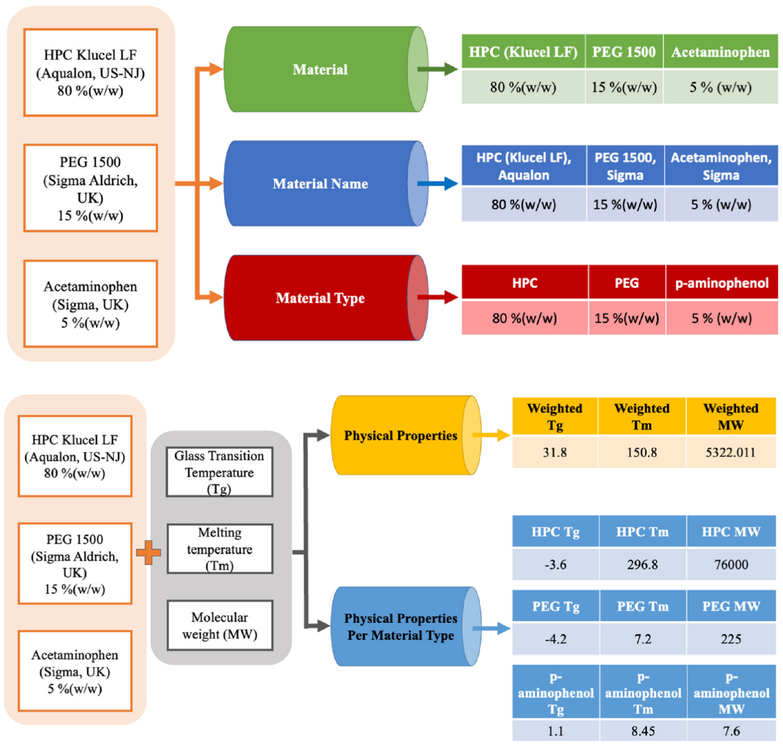

HPC Klucel LF (Aqualon, US-NJ) 80 %(w/w)
PEG 1500 (Sigma Aldrich, UK) 15 %(w/w)
Acetaminophen (Sigma, UK) 5 %(w/w)
Material
HPC (Klucel LF) | PEG 1500 | Acetaminophen
80 %(w/w) | 15 %(w/w) | 5 % (w/w)
Material Name
HPC (Klucel LF), Aqualon | PEG 1500, Sigma | Acetaminophen, Sigma
80 %(w/w) | 15 %(w/w) | 5 % (w/w)
Material Type
HPC | PEG | p-aminophenol
80 %(w/w) | 15 %(w/w) | 5 % (w/w)
Glass Transition Temperature (Tg)
Melting temperature (Tm)
Molecular weight (MW)
Physical Properties
Weighted Tg | Weighted Tm | Weighted MW
31.8 | 150.8 | 5322.011
Physical Properties Per Material Type
HPC Tg | HPC Tm | HPC MW
-3.6 | 296.8 | 76000
PEG Tg | PEG Tm | PEG MW
-4.2 | 7.2 | 225
p-aminophenol Tg | p-aminophenol Tm | p-aminophenol MW
1.1 | 8.45 | 7.6

Figure 8.3: The alternative datasets depicted. Source: [77], Figures 1 and 2.

the latest results obtained of three different ML algorithms, published in [77], contrasting the performance metrics over the alternative datasets.

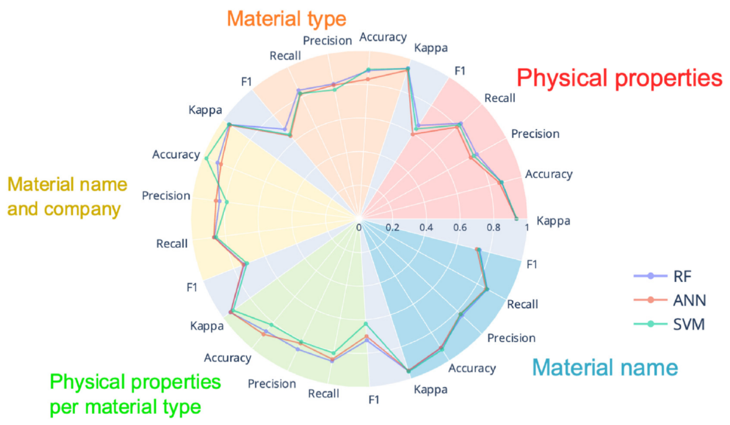


Figure 8.4: Latest ML results for three algorithms, namely MultiLayer Perceptron (MLP), Support Vector Machine (SVM) and Randon Forest (RF). Source: [77], Figure 7.

M3DISEEN Web Application

An AI-based web application, named M3DISEEN [3] [4], was designed, developed and deployed leveraging the trained ML models. With the subsequent publications, the functionality of this web application was further increased to update the models, handle new parameters and predict new target features as the dissolution profile.

8.3 aspBEEF: an ASP implementation of BEEF

BEEF: Balanced English Explanations of Forecasts

The developed tools are based on *Balanced English Explanations of Forecasts* (BEEF) [60]. The BEEF algorithm is able to learn a symbolic classification model (that we will refer to as an explainer from now on) from the predictions of an already trained machine learning classifier (hence the use of the term forecasts). The explanations obtained are said to be balanced because they may give reasons for a prediction to be in one class or in the opposite class[5]. Besides, they are able to express this explanation in natural language terms.

The aim of the algorithm is to explain the outcome of any ML binary classifier in terms of intervals over the input features. It starts from a set of predictions from the ML classifier, represented as points in a n-dimensional space where n is the number of input features of the classifier. The original method restricts itself to the binary classifier, thus, each point is labeled with 0 or 1, depending on if they are

[3] A public version of this tool is publicly available in m3diseen.com

[4] Since May 2023 the maintenance of this application is no longer the responsibility of the author of this thesis.

[5] Although the original method is originally designed for binary classification probelms, our implementation can be easily generalized for multiclass classification problems

predicted as part of the negative (respectively positive) class. Even though, our representation that will be later explained can be easily generalized for multiclass problems. In fact, the examples and the figures used to illustrate the process in this section will use a three-class problem. Over the set of predictions, BEEF first runs a traditional clustering method, such as KMeans. Then, using the traditionally found clusters as a starting point, a set of *axis-aligned, hyperrectangular clusters* (that we will refer to as *boxes*, for short) is iteratively approximated such that it maximizes the agreement in the classification with respect to the original clusters. To illustrate the method we will use 2-dimensional representations as the one shown in Figure 8.5.

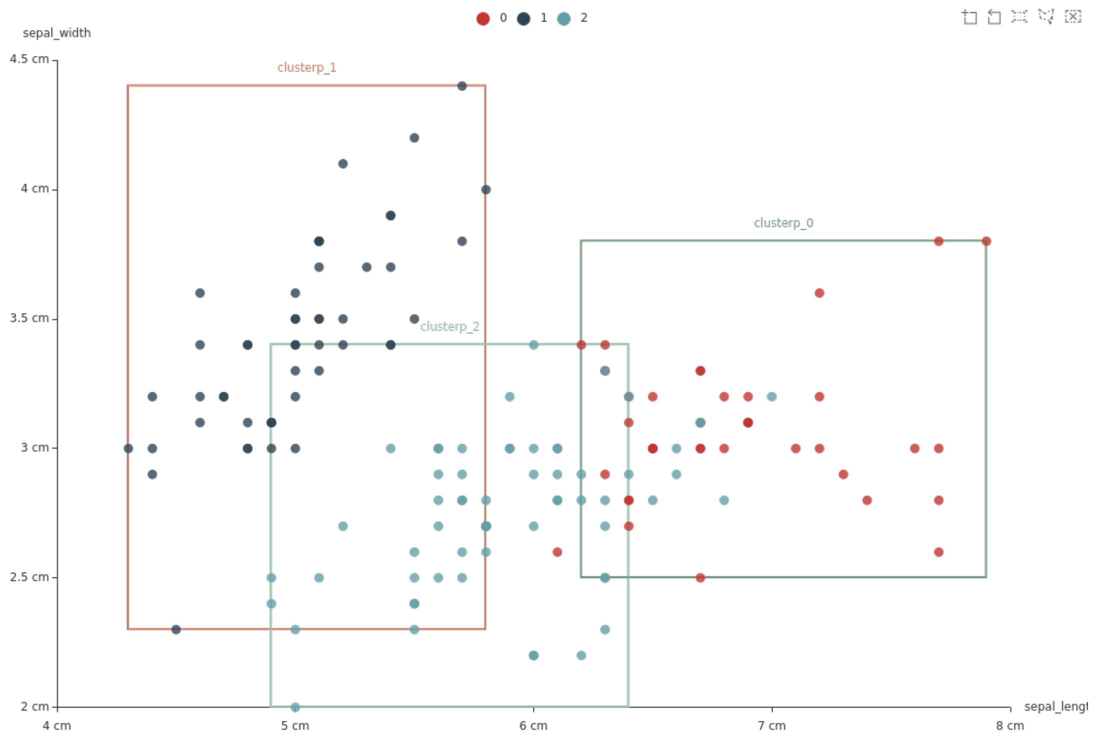


Figure 8.5: Example of some boxes approximated for the classifications of an arbitrary ML classifier in a three-class classification problem.

The problem of finding these boxes, which is named *Explanation Computation Problem*, is then stated as a combinatorial problem where a set of boxes at most h boxes have to be found such that they satisfy some predefined thresholds in terms of the following properties: *overlapping*, *purity* and *inclusion*. Firstly the *overlap* between two boxes w.r.t a given feature is defined as the amount of space shared between two boxes when projecting the boxes over the dimension of that particular feature. The property to be minimized by the algorithm though, is the *overall overlap* that is the product of the overlap between all the boxes. The second property to be minimized is called *purity* of a box, which is just the percentage that represents the population of points belonging to the majority class in that box. In the original algorithm, the particular purity of each cluster is minimized. Note that, if the number of boxes is restricted to at most h boxes, the previously defined two properties are very easily minimized by defining each box capturing only one point. Thus to avoid this they introduce the last property called *inclusion*, which is the relation of the total points that fall outside any box.

Note that in Figure 8.5, we are showing the boxes projected to two dimensions for the sake of the representation, but they are actually comprised of n dimensions. Because of that, we cannot visually know if the points fall inside the boxes or not, nor if the boxes overlap each other.

Starting from the initially found clusters, the algorithm iteratively adjusts the boundaries of the boxes, until the thresholds are satisfied. This problem has been proved to be NP-complete by the authors, their framework uses some heuristics during the search, at the cost of losing optimality.

Boxes found are *axis-aligned* and finite, so they can be described as a set of intervals over each dimension (each input feature). Besides, each box is labeled with the name of the majority predicted class within. We can explain a model outcome o just by finding out the description (the intervals) of the boxes o fell in. The descriptions of those boxes whose label matches the predicted class will be the *supporting explanations*. The rest will be the *opposing explanations*. The sum of supporting and opposing explanations is what BEEF calls a *balanced explanation*.

aspBEEF implementation

Our approach aspBEEF leverages ASP to implement a method of obtaining natural language explanations for ML classifiers. It is inspired by BEEF's combinatorial approach to finding boxes-like clusters. The main difference with respect to the original approach is that aspBEEF aims to compute an optimal set of boxes with respect to a given overlapping, impurity and inclusion thresholds, rather than using a greedy approach to find a good one.

We start from a set of predictions $p_i \in P$ from an ML classifier. Recall that each prediction is a point $p_i = (\vec{x}, y_i)$ in a n-dimensional space, where $\vec{x} = (v_1, v_2, \cdots, v_n)$, n is the number of input features of the model, and $y_i \in \{0, 1, \cdots$,c$\}$ is the class labeling the point where c is the number of classes. Then each prediction p_i is encoded as a set of atoms $value(i, f, v)$ for each $f \in \{0, 1, \cdots, n\}$ where i identifies the prediction and v [6] is the value of the prediction for the input feature f. Additionally, a $class(i, y_i)$ is introduced for any prediction p_i.

Recall also that, in BEEF, the number of allowed boxes h is determined by the user. In contrast, in aspBEEF, the number of boxes is also given by the user but it is fixed. Thus, a fact $rectangle(b)$ for each $b \in \{0, 1, \cdots, h\}$ is also introduced in the program to identify the different boxes. We will call $input(P)$ to the set of facts consisting of all the encoded predictions p_i and rectangle facts.

Let us now define the program $aspBEEF$ as the ASP program that generates a finite limit for each box b, computes its corresponding properties (i.e. overlapping, impurity and inclusion) such that the solution of program $input(P) \cup aspBEEF$ is an optimal set of boxes. We will now break down the rules in $aspBEEF$ and explain its different parts.

First, consider the following domain predicates `pid/1`, `feature/1` and `class/1` respectively modeling the set of identifiers for the predictions $p_i \in P$, the set of features $f \in \{0, 1, \ldots, n\}$ and the set of classes $cl \in \{0, 1, \cdots, c\}$ defined by the rules in Figure 8.6.

For each box $b \in \{0, 1, \cdots, h\}$, and feature $f \in \{0, 1, \ldots, n\}$ we generate the corresponding upper and lower bound of box b over the feature f. Each boundary is modeled as an atom $rectval(b, f, v)$,

[6] As the input features could be real numerical variables that cannot be handled by ASP, the values are scaled to integer number so that all decimal places are preserved.

```
pid(I)     :- value(I,A,V).
feature(A) :- value(I,A,V).
class(CL)  :- class(I,CL).
```

Figure 8.6: Some shorthand predicates for identifying predictions, features and classes (respectively).

where b is a box, f is a feature and v the position of any prediction p_i over the dimension corresponding to the feature f. The generation of the possible boundaries is done by the choice rule in Figure 8.7. As

```
2 { rectval(R,F,V) : value(_,F,V) } 2 :- rectangle(R), feature(F).
```

Figure 8.7: Choice rule generating the lower and upper bound for each box b and feature f.

the search space of possible boundaries for the boxes is infinite, we fix the possible boundaries to the position of the predictions in the space, namely the values $(v_1, v_2, \cdots, v_n) = \vec{x}$ of any prediction p_i. In Figure 8.8 we now define several rules that capture which predictions p_i fall inside the boundaries of each box b.

```
1   featureinlier(R,I,F)  :- value(I,F,V), rectval(R,F,V0), rectval(R,F,V1), V >= V0,  V <= V1, V0 < V1.
2   featureoutlier(R,I,F) :- value(I,F,V), rectval(R,F,V0), rectval(R,F,V1), V <  V0,  V0 < V1.
3   featureoutlier(R,I,F) :- value(I,F,V), rectval(R,F,V0), rectval(R,F,V1), V >  V1,  V0 < V1.
4
5   rectoutlier(R, I) :- featureoutlier(R,I,F).
6   rectinlier(R, I)  :- featureinlier(R,I,F), not rectoutlier(R,I).
7
8   outliercount(C) :- C=#count{ I : outlier(I) }.
9
10  :- rectangle(R), not rectinlier(R, _).
```

Figure 8.8: Rules from $aspBEEF$ capturing when a point `I` falls inside or outside the boundaries of box `R`.

Intuitively, predicate `featureinlier(R, I, F)` and `featureoutlier(R, I, F)` model when the position of a prediction `I` with respect to the dimension of feature `F` is between (respectively, out) of the boundaries of box `R` for feature `F`. Thus, lines 5 and 6, define when a prediction `I` falls outside (`rectoutlier(R, I)`) or inside the boundaries `rectinlier(R, I)` of box `R`. Additionally, we introduce a constraint forbidding the existence of any box `R` such that any prediction `I` falls in it.

Also, line 8 introduces a predicate `outliercount(C)` which counts the number `C` of outlier predictions. This models *exclusion* rather than inclusion as BEEF. We will thus minimize the exclusion instead.

Using these predicates, we now introduce the notion of the overlapping property in the $aspBEEF$ program using the rule in Figure 8.9. As a simplification, we just count the number of overlapping predictions that are shared by two boxes rather than computing the space shared between the boxes.

Then impurity is specified by the rules in Figure 8.10. Intuitively, `class_count(R,CL,IC)` counts the number `IC` of predictions of class `CL` falling in box `R`. Predicate `rectclass(R, CL)` models the majority class `CL` of box `R`. An impure point, represented by the predicate `impurepoint(I)`, is a prediction `I` such

```
overlappoint(I, R1, R2) :- rectinlier(R1, I), rectinlier(R2, I), R1 < R2.
overlapcount(C) :- C=#count{ I : overlappoint(I, R1, R2) }.
```

Figure 8.9: Rules finding (and counting) predictions `I` over two or more boxes.

```
class_count(R,CL,IC) :- class(CL), rectangle(R),
    IC=#count{I: class(I, CL), rectinlier(R, I)}.

rectclass(R, CL1) :- class_count(R, CL1, NCL1), not class_count(R, CL2, NCL2), NCL2 < NCL1,
    class(CL1), class(CL2), _safe(NCL2).

impurepoint(I) :- rectinlier(R,I), class(I, CLI), rectclass(R,CLR), CLI != CLR.

impurecount(C) :- C=#count{ I : impurepoint(I) }.
```

Figure 8.10: Rules modeling the class of a box `R` based on the majority class within `R`, and impure predictions `I`.

that its class does not match the box `R` it falls inside of. Finally, `impurecount(C)` counts the number `C` of total impure points with respect to the considered set of boxes. Note that, as we will minimize the overall impurity, identifying the particular boxes the impure predictions fall into makes no difference.

Finally, we use the ASP extension `asprin` [17] that allows a flexible way of defining preference relations for optimization. Figure 8.11 shows the use of `asprin`'s preferences for optimizing the set of boxes with respect to the introduced properties.

```
#const overlapprio = 3.
#const impurityprio = 2.
#const outlierprio = 1.

#preference(overlap, less(cardinality)){
    overlappoint(I, R1, R2) : rectangle(R1), rectangle(R2), pid(I)
}.
#preference(impurity, less(cardinality)){
    impurepoint(I): pid(I)
}.
#preference(outlier, less(cardinality)){
    outlier(I) : pid(I)
}.
#preference(all, lexico){
    overlapprio::**overlap; impurityprio::**impurity; outlierprio::**outlier
}.

#optimize(all).
```

Figure 8.11: `asprin` preferences for $aspBEEF$ program.

Furthermore, inspired by BEEF's[60], by Gover et al. feature selection feature, we introduced an additional option that allows the feature set to be left free to choose from. The user specifies the number of features as a constant. Figure 8.12 shows how that is specified in the $aspBEEF$ program. Of course, then the boundary selection choice rule from Figure 8.7 has to be updated to only use the chosen features (see line 4).

```
#const featurecount = 2.

featurecount { selected_feature(F): feature(F), not target(F) } featurecount.
2 { rectval(R,F,V) : value(_,F,V) } 2 :- rectangle(R), selectedfeature(F).
```

Figure 8.12: Feature selection in aspBEEF.

As a preliminary benchmark, we tested the tool finding boxes for the predictions of a ML classifier trained in random samples of three different sizes of the publicly available IRIS data set[7]. We compare performance using both the fixed and free feature selection methods. Each measure has been taken 100 times and then averaged to smooth out outlying values. The search space and the computational time grow exponentially with the number of free features. Cases in which all of the features are selected, (e.g. all of the four features in the case of Table 8.1) eliminate any decision-making over feature selection completely, thus greatly reducing the problem complexity. The best times are achieved using a pre-fixed set of features, but this requires previous knowledge of the search space.

Sample Size	Used Features	Time w/ Free Features (s)	Time w/ Fixed Features (s)
60	2	1.1288	0.7253
60	3	1.1103	0.6995
60	4	0.5443	0.5700
90	2	3.4666	1.6238
90	3	2.5741	1.3963
90	4	1.3596	1.1760
150	2	27.7299	5.5057
150	3	28.8644	5.9140
150	4	7.7072	6.1569

Table 8.1: Times table for a data set of 150 points and 4 features.

The tool was further tested for obtaining explanations for an ML that predicts the success (or failure) of the 3DP-printing process given a mixture of materials and the rest of the FDM parameters. Predictions for all experiments in the original dataset of experiments comprising a total of 614 experiments were obtained. Then, the dataset of predictions was split into random subsets of 30, 50, 100, 200, and 400 predictions. The tool was tested on all of the sub-datasets, results were collected and execution times were measured. We fixed the number of features to 2, but we left the selection of those features free. Figure 8.13 the evolution of the time invested in finding the optimal solution against the number of considered predictions. The curve shows an exponential-like evolution as the number of predictions grows. Note that for a dataset comprising 100 predictions (and using 2 free features), the time invested is around 2 minutes. After that, the invested time was (approximately) 4 hours, 145 hours and 13 days for 200, 400 and 614 predictions respectively.

Figure 8.14 shows a particular example of one of the sets of boxes found for the dataset comprising 100 predictions, and Figure 8.15 shows a couple of examples on using that boxes for obtaining expla-

[7] https://archive.ics.uci.edu/ml/datasets/iris

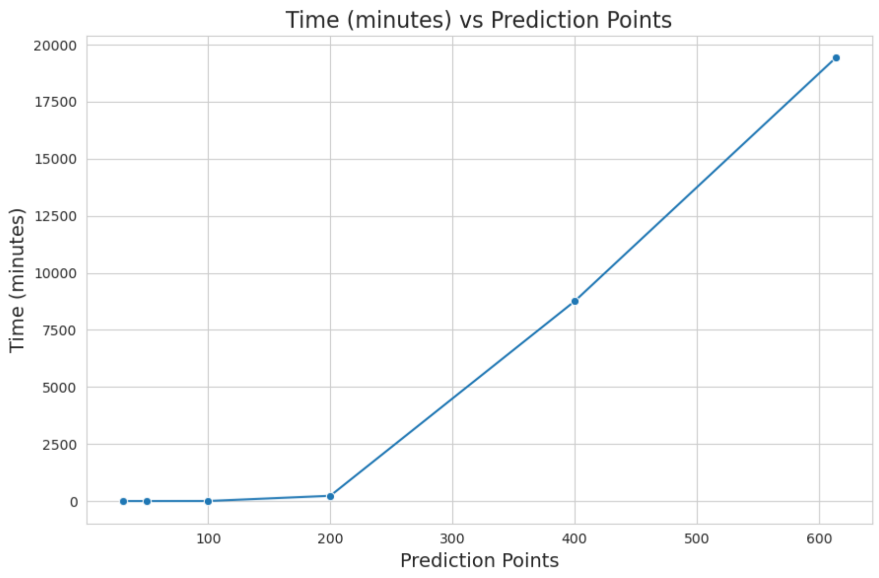


Figure 8.13: Time invested by aspBEEF in finding the optimal set of boxes (in minutes) vs the number of prediction points included in the experiment.

nations for a particular prediction. The natural language explanations were obtained by representing the boxes as an `xclingo`-annotated ASP program.

Discussion

It is worth noting that the model obtained is no longer a probabilistic model. For example, if an anomalous input is introduced, and the point falls outside all defined boxes, no prediction or explanation is possible. That is closer to human reasoning, and therefore to common sense explanations. A typical machine learning model, by contrast, always produces a prediction no matter how nonsensical the input is.

In case a point falls into multiple boxes, an explanation is generated for each box. The original BEEF explainer supports counter-explanations (hence the term Balanced). Probabilistic models usually predict the probability of the positive class (in the case of binary classification). That is, if the predicted probability is above 50% then the prediction is positive, otherwise it is negative. When the probability is close to the threshold (graphically, the prediction would fall close to the boundary that separates one class from another in the feature space) and hence the probability is very close to 50%, the model prediction is equally sharp. These cases correspond to overlapping the boxes and would produce competing explanations in BEEF. From a human perspective, balanced explanations are more intuitive.

The fact that the models obtained by BEEF are symbolic offers many possibilities. Different reasoning tasks other than inference can be performed. For example, it can be applied to counterfactual reasoning: "What should change for the prediction to be X? This opens up the possibility of generating recommendations for experiments rather than just predicting those introduced by experts.

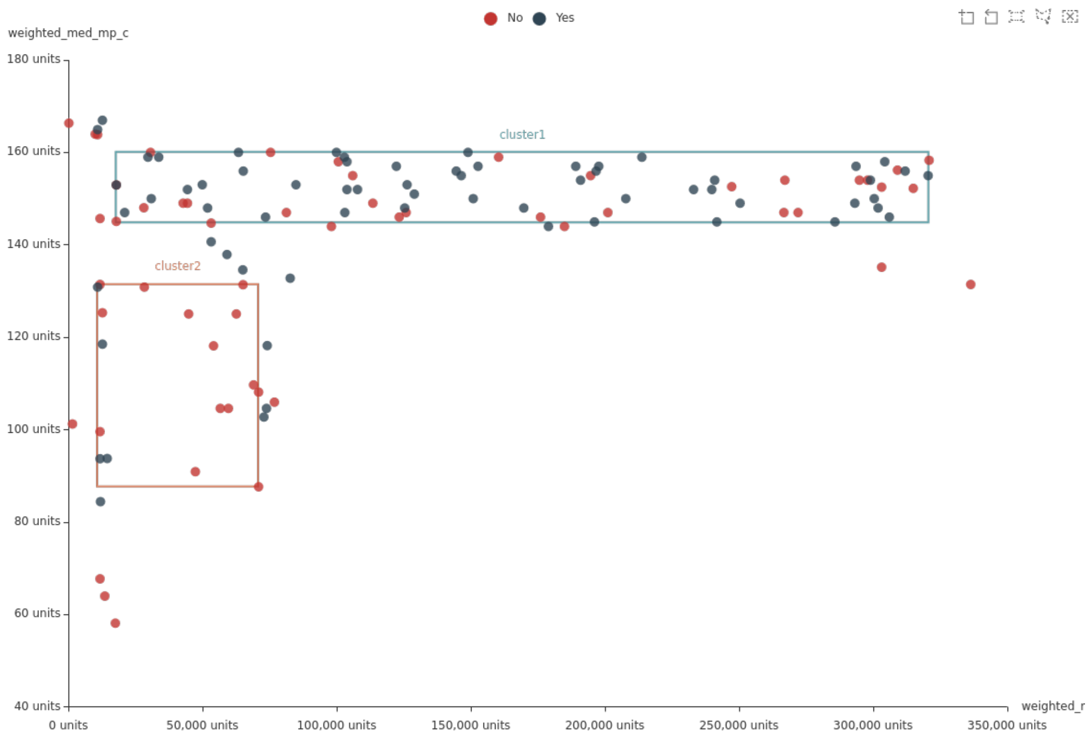


Figure 8.14: Example of a set of boxes found for the predictions of the ML model predicting the outcome of 3DP experiments.

```
*
|__The experiment is "printable" (Box 1)
|  |__Average M.Weight in [178778.3, 320548.05]  (g/
   mol)
|  |__Average M.Point in [145, 160] (<*\textdegree*>
    C)
```

```
*
|__The experiment is "not printable" (Box 2)
|  |__Average M.Weight in [10897.9, 70901.1]  (g/mol
    )
|  |__Average M.Point in [87.6, 131.4] (<*\
    textdegree*> C)
```

Figure 8.15: Explanations corresponding to predictions produced by Boxes in Figure 8.14. The left side corresponds to Box 1 whereas the right side corresponds to Box 2.

Part III

Conclusions

Chapter 9

Conclusions

In this thesis we have provided a formal characterization of explanations of logic programs in terms of graphs constructed with atoms and program rule labels called *support graphs*, as well as the notion of a justified model. Besides, we have proved that all stable models are justified whereas, in general, the opposite does not hold, at least for disjunctive programs. We have also characterized a pair of basic operations on graphs, which we call edge pruning and node forgetting, that allow performing information filtering in the explanations. We have implemented and presented a tool called `xclingo`, which computes the support graphs for ASP programs. The tool further extends the language of Answer Set Programming (ASP) with annotations that help the user design the produced explanations. We have also shown how this tool is implemented in ASP via meta-programming. We have defined the notions of commonsense explanations and technical explanations in terms of what literature on social sciences considers human-like explanations. We also discussed the process of explanation design for obtaining these kind of explanations using `xclingo`. We have shown how strongly equivalent logic programs may produce different explanations, thus leading to potentially important problems in the typical Knowledge Representation (KR) workflow. For instance, we have shown how efficient ASP encodings for solving real-world problems may not be suitable for obtaining commonsense explanations even though they are equivalent to the original program. We have provided the design of a system able to answer different types of causal questions and we have shown how this system can be implemented using `xclingo`. We have compared our approach with a relevant part of the literature on the topic of the explainability of ASP in terms of explanation definitions, practical tools, and the design of commonsense explanations. Finally, we have shown the application of two explainability techniques for obtaining explanations from different Machine Learning (ML) models for real-world problems and real-world users from outside the KR world. In particular, we have developed applications in the domains of liver transplantation and 3D printing of medicines.

The focus of the thesis has been on obtaining commonsense explanations, that is, explanations aimed at a non-expert user. For this purpose, the developed tool `xclingo` allows the use of annotations that aid in designing explanations in natural language that are user-friendly. This objective is achieved without renouncing a well-founded formalism such as support graphs, for which the tool has been proven correct by Theorems 4 and 5. This constitutes the main contribution of this thesis. Regarding the `xclingo` tool, it is easily accessible to the public and has reached a highly favorable state of usability, demonstrating very straightforward usage. In particular, the fact that it accepts the same language as `clingo` and that annotations are seen as comments makes `xclingo` a perfect companion tool to `clingo`. It can be used to obtain explanations easily without modifying the semantics of the original program. Additionally, it avoids introducing new auxiliary artifacts. Furthermore, while previous versions of the tool used an ad hoc implementation in Python, it has now been rewritten as an ASP meta-program (this is fully detailed in Section 4.3). This has proven to be very beneficial in various ways. Firstly, due to the declarative nature of ASP, it has been easier to verify that the tool is correct with respect to the definitions of support graphs. Secondly, the code is much more maintainable. Lastly, whereas the previous implementation always computed all explanations (consuming enormous computation times and becoming unviable in the worst cases), as each explanation is now a model of the ASP program, obtaining only one (or an arbitrary number n) is very straightforward. Another important contribution is the partial classification of the types of causal questions proposed in Section 5.5, which explicitly separates the notions of question and response (or explanation). Additionally, abstract proposals on how these questions could be answered are provided. While it seems that the reviewed approaches agree that positive explanations should be answered by relying in some way on some type of atom derivation, the way to respond to negative ("*Why not*") queries is what sets them apart. Although implemented differently, most rely on some form of abduction in which they imagine what should have happened (or what changes would have to be made) for the event to have occurred. In the same section, we also propose a way to implement this using `xclingo`, although the ideal scenario would be (and is indeed the most immediate future work) to have this available natively in the tool. This could be done with a new type of annotation to point out abducible atoms. In this way, when a program has no answer set, `xclingo` would automatically try to find one by applying abduction on the set of atoms that the user has declared as abducible. Including this, we also provide an abstract interactive methodology to answer the causal queries of a user in a real system, which is fully explained in a diagram in Figure 5.11. In this sense, although we did not cover all types of queries, we firmly believe that the proposed distinction constitutes an important contribution that can help clarify a taxonomy of causal questions and answers in the future.

Apart from what was mentioned earlier, we proceed to outline the areas in which this work can be extended in the future. Correspondences between support graphs and Cabalar et al's proof graphs [49] should be further studied. In Section 6.1, we outlined some of them that should naturally follow from the different notions of explanations. For instance, the possible correspondence between support graphs and $\subseteq$-minimal proof graphs. This could be further studied for other formalisms as well. Moreover, both the semantics of support graphs and the implementation of `xclingo` could be extended to capture Cabalar et al's notion of sufficient causes. Given the declarative nature of the ASP `xclingo` current implementation, this should be easy to obtain by minimizing the alternative support graphs.

Concerning new `xclingo` features, causal literals [20] could be implemented. These are literals enabling reasoning about causes in the logic programs. For instance, one could have a rule of the form:

$$a \leftarrow caused(b, c).$$

This means that we derive atom a if, in the current world, c is a cause of b. Different literals can be designed to capture different notions of causes such as direct cause, indirect cause, sufficient cause, etc. Although some of these causal literals are already possible to implement via extensions (see Section 4.4.3) other would possibly require a more complex approach. For instance, for designing a causal literal capturing the idea that *c* causes *b* in every possible explanation (or also, in every possible solution) this would possibly require the use of a multi-shot approach or the use of epistemic reasoning.

Furthermore, when we either try to abduce a counterfactual answer or simply have to select among alternative explanations, user-defined preferences could be considered. Either as an `xclingo` extension or as a new type of annotation, the user should be able to introduce those preferences in a friendly way. The use of flexible preferences like those introduced by `asprin` [17] could be studied.

Finally, we also plan to give support to sufficient causes for some types of aggregates that could be implemented as we commented on different occasions (in sections 4.2.6, 4.3.2, and 6.5). This also relates to the previous point in the sense that the intuition about which should be considered a correct cause for an aggregate is still unclear and seems to depend on the type of aggregate (and its monotonicity).

Bibliography

1. S. Ahmetaj, R. David, A. Polleres, and M. Šimkus. “Repairing SHACL Constraint Violations Using Answer Set Programming”. In: *The Semantic Web – ISWC 2022*. Ed. by U. Sattler, A. Hogan, M. Keet, V. Presutti, J. P. A. Almeida, H. Takeda, P. Monnin, G. Pirrò, and C. d’Amato. Springer International Publishing, Cham, 2022, pp. 375–391. ISBN: 978-3-031-19433-7.
2. M. Alviano, C. Dodaro, W. Faber, N. Leone, and F. Ricca. “WASP: A Native ASP Solver Based on Constraint Learning”. In: *Logic Programming and Nonmonotonic Reasoning*. Ed. by P. Cabalar and T. C. Son. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 54–66. ISBN: 978-3-642-40564-8.
3. M. Alviano, C. Dodaro, N. Leone, and F. Ricca. “Advances in WASP”. In: *Logic Programming and Nonmonotonic Reasoning*. Ed. by F. Calimeri, G. Ianni, and M. Truszczynski. Springer International Publishing, Cham, 2015, pp. 40–54. ISBN: 978-3-319-23264-5.
4. M. Alviano, L. L. T. Trieu, T. C. Son, and M. Balduccini. “Advancements in xASP, an XAI System for Answer Set Programming”. In: *Proceedings of the 38th Italian Conference on Computational Logic, Udine, Italy, June 21-23, 2023*. Ed. by A. Dovier and A. Formisano. Vol. 3428. CEUR Workshop Proceedings. CEUR-WS.org, 2023. URL: https://ceur-ws.org/Vol-3428/paper2.pdf.
5. M. Alviano, L. L. T. Trieu, T. C. Son, and M. Balduccini. “Advancements in xASP, an XAI System for Answer Set Programming”. In: *Proceedings of the 38th Italian Conference on Computational Logic, Udine, Italy, June 21-23, 2023*. Ed. by A. Dovier and A. Formisano. Vol. 3428. CEUR Workshop Proceedings. CEUR-WS.org, 2023. URL: https://ceur-ws.org/Vol-3428/paper2.pdf.
6. M. Alviano, L. L. T. Trieu, T. C. Son, and M. Balduccini. “Explanations for Answer Set Programming”. In: *Proceedings 39th International Conference on Logic Programming, ICLP 2023, Imperial College London, UK, 9th July 2023 - 15th July 2023*. Ed. by E. Pontelli, S. Costantini, C. Dodaro, S. A. Gaggl, R. Calegari, A. S. d’Avila Garcez, F. Fabiano, A. Mileo, A. Russo, and F. Toni. Vol. 385. EPTCS. 2023, pp. 27–40. DOI: 10.4204/EPTCS.385.4. URL: https://doi.org/10.4204/EPTCS.385.4.
7. J. Arias, M. Carro, Z. Chen, and G. Gupta. “Justifications for Goal-Directed Constraint Answer Set Programming”. In: *International Conference on Logic Programming, ICLP*. 2020.

8. J. Arias, M. Carro, Z. Chen, and G. Gupta. “Justifications for Goal-Directed Constraint Answer Set Programming”. In: *Proceedings 36th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2020, (Technical Communications) UNICAL, Rende (CS), Italy, 18-24th September 2020*. Ed. by F. Ricca, A. Russo, S. Greco, N. Leone, A. Artikis, G. Friedrich, P. Fodor, A. Kimmig, F. A. Lisi, M. Maratea, A. Mileo, and F. Riguzzi. Vol. 325. EPTCS. 2020, pp. 59–72. DOI: 10.4204/EPTCS.325.12. URL: https://doi.org/10.4204/EPTCS.325.12.

9. J. Arias, M. Carro, E. Salazar, K. Marple, and G. Gupta. “Constraint Answer Set Programming without Grounding”. *Theory Pract. Log. Program.* 18:3-4, 2018, pp. 337–354. DOI: 10.1017/S1471068418000285. URL: https://doi.org/10.1017/S1471068418000285.

10. J. Arias, G. Gupta, and M. Carro. “A Short Tutorial on s(CASP), a Goal-directed Execution of Constraint Answer Set Programs”. In: *Proceedings of the International Conference on Logic Programming 2021 Workshops co-located with the 37th International Conference on Logic Programming (ICLP 2021), Porto, Portugal (virtual), September 20th-21st, 2021*. Ed. by J. Arias, F. A. D’Asaro, A. Dyoub, G. Gupta, M. Hecher, E. LeBlanc, R. Peñaloza, E. Salazar, A. Saptawijaya, F. Weitkämper, and J. Zangari. Vol. 2970. CEUR Workshop Proceedings. CEUR-WS.org, 2021. URL: https://ceur-ws.org/Vol-2970/gdepaper1.pdf.

11. M. Balduccini and M. Gelfond. “Diagnostic reasoning with A-Prolog”. *Theory and Practice of Logic Programming* 3:4-5, 2003, pp. 425–461.

12. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2010. ISBN: 978-0-521-14775-0. URL: http://www.cambridge.org/de/academic/subjects/computer-science/artificial-intelligence-and-natural-language-processing/knowledge-representation-reasoning-and-declarative-problem-solving.

13. O. Bodenreider, Z. H. C, Oban, M. C. Doĝanay, E. Erdem, and H. Kosçucu. “A preliminary report on answering complex queries related to drug discovery using answer set programming.” In: *Proceedings of the Third International Workshop on Applications of Logic Programming to the (Semantic) Web and Web Services (ALPSWS 2008)*. 2008.

14. S. M. N. P. B. Bogaerts and M. Denecker. “Consistency in Justification Theory”. *NMR 2018* 20, 2018.

15. A. Bondarenko, F. Toni, and R. A. Kowalski. “An Assumption-Based Framework for Non-Monotonic Reasoning”. In: *Logic Programming and Non-monotonic Reasoning, Proceedings of the Second International Workshop, Lisbon, Portugal, June 1993*. Ed. by L. M. Pereira and A. Nerode. MIT Press, 1993, pp. 171–189.

16. M. Brain, M. Gebser, J. Pührer, T. Schaub, H. Tompits, and S. Woltran. “Debugging ASP Programs by Means of ASP”. In: *Logic Programming and Nonmonotonic Reasoning*. Ed. by C. Baral, G. Brewka, and J. Schlipf. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 31–43. ISBN: 978-3-540-72200-7.

17. G. Brewka, J. P. Delgrande, J. Romero, and T. Schaub. “asprin: Customizing Answer Set Preferences without a Headache”. In: *AAAI*. AAAI Press, 2015, pp. 1467–1474.

18. G. Brewka, T. Eiter, and M. Truszczynski. “Answer set programming at a glance”. *Communications of the ACM* 54:12, 2011, pp. 92–103.

19. P. Cabalar. "Causal Logic Programming". In: *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*. Ed. by E. Erdem, J. Lee, Y. Lierler, and D. Pearce. Vol. 7265. Lecture Notes in Computer Science. Springer, 2012, pp. 102–116. DOI: 10.1007/978-3-642-30743-0_8. URL: https://doi.org/10.1007/978-3-642-30743-0%5C_8.

20. P. Cabalar and J. Fandinno. "Enablers and inhibitors in causal justifications of logic programs". *Theory Pract. Log. Program.* 17:1, 2017, pp. 49–74. DOI: 10.1017/S1471068416000107. URL: https://doi.org/10.1017/S1471068416000107.

21. P. Cabalar, J. Fandinno, and B. Muñiz. "A System for Explainable Answer Set Programming". In: *Proc. of the 36th Intl. Conf. on Logic Programming (ICLP, Technical Communications)*. Vol. 325. EPTCS. 2020, pp. 124–136.

22. F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, and T. Schaub. "ASP-Core-2 Input Language Format". *Theory Pract. Log. Program.* 20:2, 2020, pp. 294–309. DOI: 10.1017/S1471068419000450. URL: https://doi.org/10.1017/S1471068419000450.

23. Y. Cao, S. Li, Y. Liu, Z. Yan, Y. Dai, P. Yu, and L. Sun. *A Comprehensive Survey of AI-Generated Content (AIGC): A History of Generative AI from GAN to ChatGPT*. 2023.

24. B. M. Castro, M. Elbadawi, J. J. Ong, T. Pollard, Z. Song, S. Gaisford, G. Pérez, A. W. Basit, P. Cabalar, and A. Goyanes. "Machine learning predicts 3D printing performance of over 900 drug delivery systems". *Journal of Controlled Release* 337, 2021, pp. 530–545.

25. Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, W. Ye, Y. Zhang, Y. Chang, P. S. Yu, Q. Yang, and X. Xie. "A Survey on Evaluation of Large Language Models". *ACM Trans. Intell. Syst. Technol.*, 2024. Just Accepted. ISSN: 2157-6904. DOI: 10.1145/3641289. URL: https://doi.org/10.1145/3641289.

26. K. L. Clark. "Negation as Failure". In: *Logic and Databases*. Ed. by H. Gallaire and J. Minker. Plenum, 1978, pp. 293–322.

27. E. Commision. *Artificial intelligence act*. 2021. URL: https://data.consilium.europa.eu/doc/document/ST-8115-2021-INIT/en/pdf.

28. E. Commision. *Ethics Guidelines for Trustworthy AI*. Accessed: 08-02-2024. 2019. URL: https://digital-strategy.ec.europa.eu/en/library/ethics-guidelines-trustworthy-ai.

29. K. Connor, E. O'Sullivan, L. Marson, S. Wigmore, and E. Harrison. "The Future Role of Machine Learning in Clinical Transplantation". *Transplantation* Publish Ahead of Print, 2020.

30. A. Darwiche. *Logic for Explainable AI*. 2023. arXiv: 2305.05172 [cs.AI].

31. A. Darwiche and A. Hirth. "On the (Complete) Reasons Behind Decisions". *Journal of Logic, Language and Information* 32, 2022. DOI: 10.1007/s10849-022-09377-8.

32. M. Denecker, G. Brewka, and H. Strass. "A Formal Theory of Justifications". In: *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, 2015. Proceedings*. Ed. by F. Calimeri, G. Ianni, and M. Truszczynski. Vol. 9345. Lecture Notes in Computer Science. Springer, 2015.

33. M. Denecker[1] and D. De Schreye. "Justification semantics: a unifying framework for the semantics of logic programs". In: *Logic Programming and Non-Monotonic Reasoning: Proceedings of the Second International Workshop*. MIT Press. 1993, p. 365.

34. V. Dignum. *Ethical Framework of the HumanE AI Project*. Accessed: 08-02-2024. 2019. URL: https://www.humane-ai.eu/wp-content/uploads/2019/11/D13-HumaneAI-framework-report.pdf.

35. C. Dodaro, P. Gasteiger, B. Musitsch, F. Ricca, and K. Shchekotykhin. "Interactive Debugging of Non-ground ASP Programs". In: *Logic Programming and Nonmonotonic Reasoning*. Ed. by F. Calimeri, G. Ianni, and M. Truszczynski. Springer International Publishing, Cham, 2015, pp. 279–293. ISBN: 978-3-319-23264-5.

36. P. Dung, R. Kowalski, and F. Toni. "Assumption-Based Argumentation". *Argumentation in Artificial Intelligence*, 2009, pp. 199–218.

37. P. Dutkowski, C. E. Oberkofler, K. Slankamenac, M. A. Puhan, E. Schadde, B. Müllhaupt, A. Geier, and P. A. Clavien. "Are there better guidelines for allocation in liver transplantation?: A novel score targeting justice and utility in the model for end-stage liver disease era," *Ann. Surg* 254:5, 2011, pp. 745–754.

38. T. Eiter and T. Geibinger. "Explaining Answer-Set Programs with Abstract Constraint Atoms". In: *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*. ijcai.org, 2023, pp. 3193–3202. DOI: 10.24963/IJCAI.2023/356. URL: https://doi.org/10.24963/ijcai.2023/356.

39. T. Eiter, T. Geibinger, and J. Oetsch. "Contrastive Explanations for Answer-Set Programs". In: *Logics in Artificial Intelligence - 18th European Conference, JELIA 2023, Dresden, Germany, September 20-22, 2023, Proceedings*. Ed. by S. A. Gaggl, M. V. Martinez, and M. Ortiz. Vol. 14281. Lecture Notes in Computer Science. Springer, 2023, pp. 73–89. DOI: 10.1007/978-3-031-43619-2_6. URL: https://doi.org/10.1007/978-3-031-43619-2%5C_6.

40. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. "Effective Integration of Declarative Rules with External Evaluations for Semantic-Web Reasoning". In: *The Semantic Web: Research and Applications, 3rd European Semantic Web Conference, ESWC 2006, Budva, Montenegro, June 11-14, 2006, Proceedings*. Ed. by Y. Sure and J. Domingue. Vol. 4011. Lecture Notes in Computer Science. Springer, 2006, pp. 273–287. DOI: 10.1007/11762256_22. URL: https://doi.org/10.1007/11762256%5C_22.

41. M. Elbadawi, B. M. Castro, F. K. Gavins, J. J. Ong, S. Gaisford, G. Pérez, A. W. Basit, P. Cabalar, and A. Goyanes. "M3DISEEN: A novel machine learning approach for predicting the 3D printability of medicines". *International Journal of Pharmaceutics* 590, 2020, p. 119837.

42. M. H. van Emden and R. A. Kowalski. "The Semantics of Predicate Logic as a Programming Language". *Journal of the ACM* 23, 1976, pp. 733–742.

43. E. Erdem, Y. Erdem, H. Erdogan, and U. Öztok. "Finding Answers and Generating Explanations for Complex Biomedical Queries". In: *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. Ed. by W. Burgard and D. Roth. AAAI Press, 2011, pp. 785–790. DOI: 10.1609/AAAI.V25I1.7946. URL: https://doi.org/10.1609/aaai.v25i1.7946.

44. E. Erdem, M. Gelfond, and N. Leone. “Applications of Answer Set Programming”. *AI Mag.* 37:3, 2016, pp. 53–68. DOI: 10.1609/AIMAG.V37I3.2678. URL: https://doi.org/10.1609/aimag.v37i3.2678.

45. E. Erdem and U. Öztok. “Generating explanations for biomedical queries”. *Theory Pract. Log. Program.* 15:1, 2015, pp. 35–78.

46. E. Erdem and R. Yeniterzi. “Transforming Controlled Natural Language Biomedical Queries into Answer Set Programs”. In: *Proceedings of the BioNLP Workshop, BioNLP@HLT-NAACL 2009, Boulder, Colorado, USA, June 4-5, 2009.* Ed. by K. B. Cohen, D. Demner-Fushman, S. Ananiadou, J. Pestian, J. Tsujii, and B. L. Webber. Association for Computational Linguistics, 2009, pp. 117–124. URL: https://aclanthology.org/W09-1315/.

47. J. Fandinno. “A Causal Semantics for Logic Programming”. PhD thesis. Facultad de Informática, University of A Coruña, 2015.

48. J. Fandinno and C. Schulz. “Answering the "why" in answer set programming - A survey of explanation approaches”. *Theory and Practice of Logic Programming* 19:2, 2019, pp. 114–203.

49. J. Fandiño. “A Causal Semantics for Logic Programming”. PhD thesis. University of Coruña, 2015.

50. P. Gasteiger, C. Dodaro, B. Musitsch, K. Reale, F. Ricca, and K. Schekotihin. “An integrated Graphical User Interface for Debugging Answer Set Programs”. *CoRR* abs/1611.04969, 2016. arXiv: 1611.04969. URL: http://arxiv.org/abs/1611.04969.

51. M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and P. Wanko. “Theory Solving Made Easy with Clingo 5”. In: *32nd Intl. Conf. on Logic Programming (ICLP, Technical Communications).* Ed. by M. Carro and A. King. Vol. 52. OASIcs. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 2:1–2:15.

52. M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. “Clingo = ASP + Control: Preliminary Report”. *ArXiv* abs/1405.3694, 2014. URL: https://api.semanticscholar.org/CorpusID:7977544.

53. M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and P. Wanko. “Theory Solving Made Easy with Clingo 5”. In: *Technical Communications of the 32nd International Conference on Logic Programming, ICLP 2016 TCs, October 16-21, 2016, New York City, USA.* Ed. by M. Carro, A. King, N. Saeedloei, and M. D. Vos. Vol. 52. OASIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 2:1–2:15. DOI: 10.4230/OASICS.ICLP.2016.2. URL: https://doi.org/10.4230/OASIcs.ICLP.2016.2.

54. M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012. ISBN: 978-3-031-00433-9. DOI: 10.2200/S00457ED1V01Y201211AIM019. URL: https://doi.org/10.2200/S00457ED1V01Y201211AIM019.

55. M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. “Multi-shot ASP solving with clingo”. *CoRR* abs/1705.09811, 2017.

56. M. Gebser, J. Pührer, T. Schaub, and H. Tompits. “A Meta-Programming Technique for Debugging Answer-Set Programs.” In: vol. 1. 2008, pp. 448–453.

57. M. Gelfond and V. Lifschitz. “The stable models semantics for logic programming”. In: *Proc. of the 5th International Conference on Logic Programming*. 1988, pp. 1070–1080.

58. M. Ghassemi, L. Oakden-Rayner, and A. L. Beam. “The false hope of current approaches to explainable artificial intelligence in health care”. *The Lancet Digital Health* 3:11, 2021, e745–e750. ISSN: 2589-7500. DOI: https://doi.org/10.1016/S2589-7500(21)00208-9. URL: https://www.sciencedirect.com/science/article/pii/S2589750021002089.

59. A. Goyanes, F. Fina, A. Martorana, D. Sedough, S. Gaisford, and A. Basit. “Development of modified release 3D printed tablets (printlets) with pharmaceutical excipients using additive manufacturing”. *International Journal of Pharmaceutics* 527, 2017. DOI: 10.1016/j.ijpharm.2017.05.021.

60. S. Grover, C. Pulice, G. I. Simari, and V. S. Subrahmanian. “BEEF: Balanced English Explanations of Forecasts”. *IEEE Transactions on Computational Social Systems* 6:2, 2019, pp. 350–364. DOI: 10.1109/TCSS.2019.2902490. URL: https://doi.org/10.1109/TCSS.2019.2902490.

61. D. Guijo-Rubio, J. Briceño, P. A. Gutiérrez, M. Ayllón, R. Ciria, and C. Martínez. “Statistical methods versus machine learning techniques for donor-recipient matching in liver transplantation”. *PloS one* 16, 2021, e0252068. DOI: 10.1371/journal.pone.0252068.

62. N. R. Hanson. *Patterns of Discovery*. University Press, Cambridge [Eng.], 1958.

63. K. B. Klein, T. D. Stafinski, and D. Menon. “Predicting survival after liver transplantation based on pre-transplant MELD score: A systematic review of the literature”. *PLoS One* 8:12, 2013, e80661.

64. R. A. Kowalski, J. A. Dávila, G. Sartor, and M. Calejo. “Logical English for Law and Education”. In: *Prolog: The Next 50 Years*. Ed. by D. S. Warren, V. Dahl, T. Eiter, M. V. Hermenegildo, R. A. Kowalski, and F. Rossi. Vol. 13900. Lecture Notes in Computer Science. Springer, 2023, pp. 287–299. DOI: 10.1007/978-3-031-35254-6_24. URL: https://doi.org/10.1007/978-3-031-35254-6%5C_24.

65. E. A. Lee. *Limits of Machines, Limits of Humans*. 2022.

66. V. Lifschitz, D. Pearce, and A. Valverde. “Strongly equivalent logic programs”. *ACM Trans. Comput. Log.* 2, 2001, pp. 526–541. DOI: 10.1145/502166.502170.

67. C.-L. Liu, R.-S. Soong, W.-C. Lee, G.-W. Jiang, and Y.-C. Lin. “Predicting Short-term Survival after Liver Transplantation using Machine Learning”. *Scientific Reports* 10, 2020. DOI: 10.1038/s41598-020-62387-z.

68. S. Lundberg and S.-I. Lee. “A Unified Approach to Interpreting Model Predictions”. In: 2017.

69. V. W. Marek, I. Niemelä, and M. Truszczynski. “Logic programs with monotone abstract constraint atoms”. *Theory Pract. Log. Program.* 8:2, 2008, pp. 167–199. DOI: 10.1017/S147106840700302X. URL: https://doi.org/10.1017/S147106840700302X.

70. K. Marple, E. Salazar, and G. Gupta. “Computing Stable Models of Normal Logic Programs Without Grounding”. *CoRR* abs/1709.00501, 2017. arXiv: 1709.00501. URL: http://arxiv.org/abs/1709.00501.

71. S. Marynissen. “Advances in Justification Theory”. PhD thesis. Ph. D. thesis, Department of Computer Science, KU Leuven. Denecker, Marc and ..., 2022.

72. S. Marynissen and B. Bogaerts. “Tree-Like Justification Systems are Consistent”. *arXiv preprint arXiv:2208.03089*, 2022.

73. J. McCarthy. “Programs with common sense”. In: *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*. 1959, pp. 75–91.

74. D. V. McDermott and J. Doyle. “Non-Monotonic Logic I”. *Artificial Intelligence* 13:1-2, 1980, pp. 41–72.

75. T. Miller. “Explanation in artificial intelligence: Insights from the social sciences”. *Artificial Intelligence* 267, 2019, pp. 1–38.

76. J. Oetsch, J. Pührer, and H. Tompits. “Catching the Ouroboros: On Debugging Non-ground Answer-Set Programs”. *Theory and Practice of Logic Programming* 10, 2010. DOI: 10.1017/S1471068410000256.

77. J. J. Ong, B. M. Castro, S. Gaisford, P. Cabalar, A. W. Basit, G. Pérez, and A. Goyanes. “Accelerating 3D printing of pharmaceutical products using machine learning”. *International Journal of Pharmaceutics: X* 4, 2022, p. 100120.

78. U. Öztok and E. Erdem. “Generating Explanations for Complex Biomedical Queries”. In: *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. Ed. by W. Burgard and D. Roth. AAAI Press, 2011, pp. 1806–1807. DOI: 10.1609/AAAI.V25I1.8055. URL: https://doi.org/10.1609/aaai.v25i1.8055.

79. D. Pearce. “A New Logical Characterisation of Stable Models and Answer Sets”. In: *Non monotonic extensions of logic programming. Proc. NMELP’96. (LNAI 1216)*. Springer-Verlag, 1997.

80. J. Pearl. *Causality*. 2$^{\text{nd}}$ ed. Cambridge University Press, 2009. DOI: 10.1017/CBO9780511803161.

81. J. Pearl. “Reasoning with Cause and Effect”. In: *Proc. of the 16th International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden*. Ed. by T. Dean. Morgan Kaufmann, 1999.

82. J. Pearl and D. Mackenzie. *The Book of Why. The New Science of Cause and Effect*. Basic Books, New York, 2018. ISBN: 978-0-465-09760-9.

83. A. Pieper. “Advanced Topic in Interactive Product Configuration”. Bachelor’s thesis. 2023.

84. E. Pontelli and T. C. Son. “*Justifications* for Logic Programs Under Answer Set Semantics”. In: *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Ed. by S. Etalle and M. Truszczynski. Vol. 4079. Lecture Notes in Computer Science. Springer, 2006, pp. 196–210. DOI: 10.1007/11799573_16. URL: https://doi.org/10.1007/11799573%5C_16.

85. E. Pontelli, T. C. Son, and O. El-Khatib. “Justifications for logic programs under answer set semantics”. *Theory Pract. Log. Program.* 9:1, 2009, pp. 1–56. DOI: 10.1017/S1471068408003633. URL: https://doi.org/10.1017/S1471068408003633.

86. A. Rana, M. A. Hardy, K. J. Halazun, D. C. Woodland, L. E. Ratner, B. Samstein, J. V. Guarrera, R. S. Brown Jr, and J. C. Emond. "Survival Outcomes Following Liver Transplantation (SOFT) Score: A Novel Method to Predict Patient Survival Following Liver Transplantation". *American Journal of Transplantation* 8:12, 2008, pp. 2537–2546.

87. R. Reiter. "A theory of diagnosis from first principles". *Artificial Intelligence* 32:1, 1987, pp. 57–95. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(87)90062-2. URL: https://www.sciencedirect.com/science/article/pii/0004370287900622.

88. M. Ribeiro, S. Singh, and C. Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier". In: 2016, pp. 97–101. DOI: 10.18653/v1/N16-3020.

89. Z. G. Saribatur and T. Eiter. "Omission-Based Abstraction for Answer Set Programs". *Theory Pract. Log. Program.* 21:2, 2021, pp. 145–195. DOI: 10.1017/S1471068420000095. URL: https://doi.org/10.1017/S1471068420000095.

90. Z. G. Saribatur, T. Eiter, and P. Schüller. "Abstraction for non-ground answer set programs". *Artif. Intell.* 300, 2021, p. 103563. DOI: 10.1016/J.ARTINT.2021.103563. URL: https://doi.org/10.1016/j.artint.2021.103563.

91. G. Sartor, J. A. Dávila, M. Billi, G. Contissa, G. Pisano, and R. A. Kowalski. "Integration of Logical English and s(CASP)". In: *Proceedings of the International Conference on Logic Programming 2022 Workshops co-located with the 38th International Conference on Logic Programming (ICLP 2022), Haifa, Israel, July 31st - August 1st, 2022*. Ed. by J. Arias, R. Calegari, L. Dickens, W. Faber, J. Fandinno, G. Gupta, M. Hecher, D. Inclezan, E. LeBlanc, M. Morak, E. Salazar, and J. Zangari. Vol. 3193. CEUR Workshop Proceedings. CEUR-WS.org, 2022. URL: https://ceur-ws.org/Vol-3193/paper5GDE.pdf.

92. K. Schekotihin. "Interactive Query-Based Debugging of ASP Programs". *Proceedings of the AAAI Conference on Artificial Intelligence* 29, 2015. DOI: 10.1609/aaai.v29i1.9394.

93. C. Schulz. "Argumentation for Answer Set Programming and other Non-monotonic Reasoning Systems". *Theory Pract. Log. Program.* 13:4-5-Online-Supplement, 2013. URL: http://static.cambridge.org/resource/id/urn:cambridge.org:id:binary:20161018085635834-0697:S1471068413000112:tlp2013038.pdf.

94. C. Schulz and F. Toni. "ABA-Based Answer Set Justification". *Theory Pract. Log. Program.* 13:4-5-Online-Supplement, 2013. URL: http://static.cambridge.org/resource/id/urn:cambridge.org:id:binary:20161018085635834-0697:S1471068413000112:tlp2013002.pdf.

95. C. Schulz and F. Toni. "Justifying answer sets using argumentation". *Theory Pract. Log. Program.* 16:1, 2016, pp. 59–110. DOI: 10.1017/S1471068414000702. URL: https://doi.org/10.1017/S1471068414000702.

96. C. Schulz and F. Toni. "Logic Programming in Assumption-Based Argumentation Revisited - Semantics and Graphical Representation". In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. Ed. by B. Bonet and S. Koenig. AAAI Press, 2015, pp. 1569–1575. DOI: 10.1609/AAAI.V29I1.9417. URL: https://doi.org/10.1609/aaai.v29i1.9417.

97. T. Syrjänen. “Debugging inconsistent answer set programs”. In: *Proceedings of the 11th International Workshop on Non-Monotonic Reasoning (NMR'06)*. 2006, pp. 77–84.

98. S. J. Trenfield, A. Awad, A. Goyanes, S. Gaisford, and A. W. Basit. “3D Printing Pharmaceuticals: Drug Development to Frontline Care”. *Trends in Pharmacological Sciences* 39:5, 2018, pp. 440–451. ISSN: 0165-6147. DOI: https://doi.org/10.1016/j.tips.2018.02.006. URL: https://www.sciencedirect.com/science/article/pii/S0165614718300440.

99. L. L. Trieu, T. C. Son, and M. Balduccini. “xASP: An Explanation Generation System for Answer Set Programming”. In: *Logic Programming and Nonmonotonic Reasoning*. Ed. by G. Gottlob, D. Inclezan, and M. Maratea. Springer International Publishing, Cham, 2022, pp. 363–369. ISBN: 978-3-031-15707-3.

100. L. L. Trieu, T. C. Son, E. Pontelli, and M. Balduccini. “Generating explanations for answer set programming applications”. In: *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications III*. Ed. by T. Pham and L. Solomon. Vol. 11746. International Society for Optics and Photonics. SPIE, 2021, p. 117461L. DOI: 10.1117/12.2587517. URL: https://doi.org/10.1117/12.2587517.

101. L. L. T. Trieu, T. C. Son, and M. Balduccini. “exp(ASPc) : Explaining ASP Programs with Choice Atoms and Constraint Rules”. In: *Proceedings 37th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2021, Porto (virtual event), 20-27th September 2021*. Ed. by A. Formisano, Y. A. Liu, B. Bogaerts, A. Brik, V. Dahl, C. Dodaro, P. Fodor, G. L. Pozzato, J. Vennekens, and N. Zhou. Vol. 345. EPTCS. 2021, pp. 155–161. DOI: 10.4204/EPTCS.345.28. URL: https://doi.org/10.4204/EPTCS.345.28.

102. L. L. T. Trieu, T. C. Son, and M. Balduccini. “xASP: An Explanation Generation System for Answer Set Programming”. In: *Logic Programming and Nonmonotonic Reasoning - 16th International Conference, LPNMR 2022, Genova, Italy, September 5-9, 2022, Proceedings*. Ed. by G. Gottlob, D. Inclezan, and M. Maratea. Vol. 13416. Lecture Notes in Computer Science. Springer, 2022, pp. 363–369. DOI: 10.1007/978-3-031-15707-3_28. URL: https://doi.org/10.1007/978-3-031-15707-3%5C_28.

103. H. Weichelt. “Finding minimal unsatisfiable cores”. Bachelor's thesis. 2023.

104. R. Wiesner, E. Edwards, R. Freeman, A. Harper, R. Kim, P. Kamath, W. Kremers, J. Lake, T. Howard, R. M. Merion, and et al. “Model for end-stage liver disease (MELD) and allocation of donor livers”. *Gastroenterology* 124:1, 2003, pp. 91–96.

105. T. Zhu, Z. Zhang, Y. Zhai, and Z. Gao. “A processing method for inconsistent answer set programs based on minimal principle”. In: *International Conference on Automatic Control and Artificial Intelligence (ACAI 2012)*. 2012, pp. 270–274. DOI: 10.1049/cp.2012.0971.

Appendix A

Markup Annotations in Detail

`xclingo` is fed with standard ASP programs which can be extended with so-called *annotations*. The currently defined markup annotation language of `xclingo` includes up to 5 different kinds of annotations which mainly support 2 purposes. On one hand, they are used by the user to define what has to be explained, which is the case of the `show_trace` annotation. On the other hand, the annotations help in designing the final explanations. The user can make use of `trace` and `trace_rule` annotations to define the relevant information that will take part in the explanations but to customize the text that will be displayed as well. By contrast, `mute` and `mute_body` annotations are used to remove non-causal aspects of ASP programs that we normally do not want to include in the explanations.

All kinds of annotations start with a particular token (for instance, `show_trace` for the *Show trace* annotation), and all the tokens start with the standard comment ASP symbol `%`. This means that any of the annotations introduced by `xclingo`'s markup annotation language do not modify the semantics of the extended ASP program at all. After that, each annotations have its own particular syntax. Figure A.1 shows the complete BNF of the `xclingo`'s markup annotation language.

```
<list-of-variables> ::= <ASP-variable> ["," <list-of-variables>]

<showtrace> ::= "%!show\_trace" "{" <ASP-non-ground-atom> "}" ":-" <ASP-rule-body> "."
<trace> ::= "%!trace" "{" <ASP-non-ground-atom> "," <ASP-string> "," <list-of-variables> "}" ":-" <ASP-rule-body> "."
<mute> ::= "%!mute" "{" <ASP-non-ground-atom> "}" ":-" <ASP-rule-body> "."
<tracerule> ::= "%!trace_rule" "{" <ASP-string> "," <list-of-variables> "}" "."
<muterule> ::= "%!mute_rule" "."
```

Listing A.1: Complete BNF for `xclingo`'s markup annotation langugae

The BNF rules `<ASP-variable>`, `<ASP-non-ground-atom>`, `<ASP-rule-body>` and `<ASP-string>` follow the standard syntax for the corresponding ASP standard element. The rules `<tracerule>` and `<muterule>` must be placed right before the rules that will be affected by the annotation.

The rest of the Appendix visits the particular usage and syntax of each annotation in detail, as well as gives some advice on their application and/or the organization of annotations in files. The particular implementation of each one is discussed in Section 4.3.

A.1 Show Trace annotation

The *Show trace* annotation is used to define the set of atoms that `xclingo` will explain. In that sense, this annotation acts as a query that the user writes for the set of atoms that she wants to obtain an explanation for.

The BNF for the syntax of the `show_trace` annotation is shown in Listing A.2.

```
<showtrace> ::= "%!show\_trace" "{" <ASP-non-ground-atom> "}" ":-" <ASP-rule-body> "."
```

Listing A.2: BNF for the `show_trace` annotation

As it can be seen, the syntax following the `show_trace` starting token recalls a standard ASP rule, where the `<ASP-non-ground-atom>` between the curly braces would act as a non-disjunctive head for the rule. Any atom derived from this rule would be considered a *queried atom*, this is, as part of the set of atoms that the user wants `xclingo` to explain. Any queried atom that is true in the answer set is then explained. Of course, several `show_trace` annotations can be included when running a program and all of them contribute to the set of queried atoms.

It can be interesting to maintain your `show_trace` sentences in a separate file. This file would act as the query to `xclingo`, in a way we could have multiple files that feature different queries. This *query files* can be mixed, replaced or even reused for different ASP encodings.

A.2 Trace annotation

By default, `xclingo` considers any atom in the answer set as a *forgettable* atom and thus, they will be removed from the final explanations. To prevent this default behavior for a set of atoms, the user can write *Trace* annotations. Each *Trace* annotation defines a set of atoms that will be considered as *traced* (*not forgettable*) and therefore will be part of the tree explanations. In addition, it also allows the user to define parametrized text labels that replace the atoms when displaying the explanations.

The BNF for the syntax of the `trace` annotation is shown in Listing A.3.

```
<list-of-variables> ::= <ASP-variable> ["," <list-of-variables>]
<trace> ::= "%!trace" "{" <ASP-non-ground-atom> "," <ASP-string> "," <list-of-variables> "}" ":-" <ASP-rule-body> "."
```

Listing A.3: BNF for the `trace` annotation

If compared with the `show_trace` annotation syntax, we now see a couple of additional elements. In particular, the `<ASP-string>` and `<list-of-variables>` after the `<ASP-non-ground-atom>`. As the `show_trace` annotation, we can imagine that it works as a standard ASP rule, from where derived atoms will be considered as *traced* atoms. Additionally, for each derived atom a particular text string will also be derived and associated with its corresponding derived atom. This text will be used later for replacing the atoms in the final tree explanations.

The `<ASP-string>` element is a parametrized string. Inside the given text, the user can make use of several placeholders `%` to link the values of the variables referenced in the `<list-of-variables>`, that must have been used in `<ASP-rule-body>`. The linkage is done by the order the variables appear in the list. For instance, in the following annotation

```
%!trace {values(VAR1,VAR2,VAR3) , "Values: VAR1=%  VAR2=%  VAR3=%", VAR1, VAR2, VAR3} :- values(VAR1,VAR2,VAR3).
```

the first `%` placeholder will take the value of variable `VAR1`, the second will take it from `VAR2` and the last from `VAR3`. The number of placeholders and referenced variables is not limited.

A.3 Mute annotation

In terms of syntax, `mute` annotations work in the same way `show_trace` annotation does. Listing A.4 shows the BNF for the `mute` annotation.

```
<mute> ::= "%!mute" "{" <ASP-non-ground-atom> "}" ":-" <ASP-rule-body> "."
```

Listing A.4: BNF for the `mute` annotation

It acts as a standard ASP rule and the derived atoms will be treated as *muted atoms*. These atoms can't act as a cause for any other atom, this is, any cause-effect relation whereas a *muted atom* is a cause that will be removed from the explanation.

The most convenient and typical application of this annotation is to remove non-causal relations that are assumed by `xclingo` from our ASP code. For instance, often some domain predicates are derived from others like in the following rule:

```
person(S) :- student(S).
```

However, understanding this rule as causal (i.e. *S is a person because is a student*) would be a mistake. This default behavior can be prevented by *muting* the atom `person`.

From a practical perspective, this annotation is used to remove edges from the explanation graph. More precisely, the outgoing edges of *muted atoms* are removed, just as it is explained in the Definition 9 of the *edge prunning* operation.

As it was demonstrated in Section 4.2.4, the impact of *muting* non-causal atoms in the final explanations is often critical. We, therefore, recommend always bearing in mind muting this type of relations to prevent an explosion in the number of explanations.

A.4 Trace_rule annotation

In Chapter 3, we have seen how when several rules support an atom for a particular Answer Set we create different explanation graphs. In each of these graphs, each atom is only *labeled* by one of the rules. However, `trace` and `mute` annotations allow us to manipulate the design of the final explanation at an *atom level*. This is, the user can control which atoms appear (or do not appear) in the explanations and which atoms should act as causes (or not) disregarding which rules these atoms were derived from. So if what we need is to differentiate which rule an atom was derived from, we can not count on them. For that purpose, we have *Trace rule* annotations. This kind of annotation works similarly to `trace` annotations, but they work on a *rule level*. This is, they mark atoms as *traced* and link text labels to them, but only to the atoms in the current explanation graph that share its label with a particular rule.

This annotation must be written preceding the ASP rule to which is to apply. In terms of the syntax, it is simpler when compared with previously shown annotations. The BNF of the `trace_rule` annotation is shown in Listing A.5.

```
<tracerule> ::= "%!trace_rule" "{" <ASP-string> "," <list-of-variables> "}" "."
```

Listing A.5: BNF for the `trace_rule` annotation

Since the set of atoms is determined by the corresponding *traced rule*, in this annotation we do not find any syntax for that. Indeed, we only have the needed element to create the text labels that the atoms will be replaced by. The customization of the text works in the same manner it does in the case of `trace` annotations. The placeholders in the text are orderly replaced by the variables in the list, which take the values from the body of the corresponding *traced rule*.

A.5 Mute_rule annotations

The usage is very similar to `trace_rule` annotation's usage. The user only has to write the `mute_body` token preceding the rule that is meant to mute. Listing A.6 shows its BNF.

```
<muterule> ::= "%!mute_rule" "."
```

Listing A.6: BNF for the `mute_body` annotation

As it does not have any other extra parameter, its syntax remains very simple.

In essence, it works as a *rule level mute annotation*. When used, this kind of annotation targets a rule and marks it as *Muted*. Any atom derived from a *muted* rule will act as if it has no causes. From a practical perspective is a way to remove edges from the explanation graph. More precisely, the incoming edges of atoms *labeled* with *muted rule*'s labels are removed, just as it is explained in the Definition 9 of the *edge prunning* operation.

Appendix B

Xcligo's logic program

```
%%%%%%%%%%%%%% xclingo_fired.lp %%%%%%%%%%%%%%%%%

% Marks relevant atoms of the program, with respect of the atoms that must be explained.
_xclingo_relevant(ToExplainAtom) :- _xclingo_show_trace(ToExplainAtom).
_xclingo_relevant(Cause) :- _xclingo_relevant(Effect), _xclingo_depends(_xclingo_sup(R, D, Effect, Vars), Cause).

%%%% fireable if it fact
_xclingo_fbody(RuleID, D, Atom, Vars) :- _xclingo_relevant(Atom), _xclingo_sup(RuleID, D, Atom, Vars), not _xclingo_depends(
      _xclingo_sup(RuleID, D, _, _), _).
%%%% _xclingo_fbody if
_xclingo_fbody(R, D, Atom, Vars) :-
    _xclingo_sup(R, D, Atom, Vars),
    _xclingo_f_atom(Cause) : _xclingo_depends(_xclingo_sup(R, D, Atom, Vars), Cause).

% Decides which rule fire each relevant atom (must be one and only one).
1{_xclingo_f(RuleID, D, Atom, Vars) : _xclingo_fbody(RuleID, D, Atom, Vars)}1 :- _xclingo_relevant(Atom).
% Two elements from the same disyunction cannot be selected
:- _xclingo_f(R,D1,_,_), _xclingo_f(R,D2,_,_), D1!=D2.


_xclingo_f_atom(Atom) :- _xclingo_f(_, _, Atom, _).

_xclingo_direct_cause(RuleID, Effect, Cause) :- _xclingo_f(RuleID, DisID, Effect, Vars), _xclingo_depends(_xclingo_sup(
      RuleID, DisID, Effect, Vars), Cause).
```

Listing B.1: Complete contents of logic program *xclingo_fired.lp*.

```
%%%%%%%%%%%%%% xclingo_graph.lp %%%%%%%%%%%%%%%%%

% Complete explanation graph (handles %!mute) (includes non labelled atoms)
_xclingo_graph(complete_explanation).
_xclingo_node(ToExplainAtom, complete_explanation) :- _xclingo_show_trace(ToExplainAtom), not _xclingo_muted(ToExplainAtom).
_xclingo_edge((Caused, Cause), complete_explanation) :-
    _xclingo_node(Caused, complete_explanation),
    _xclingo_direct_cause(RuleID, Caused, Cause),
    not _xclingo_muted(Cause),
    not _xclingo_muted_body(RuleID).
_xclingo_node(Atom, complete_explanation) :- _xclingo_edge((_, Atom), complete_explanation).

% Compressing graph (only labelled; and show_trace Atoms even if they are not labelled)
_xclingo_visible(X) :- _xclingo_node(X, complete_explanation), _xclingo_label(X, _).
%
_xclingo_skip(X, Y) :- _xclingo_edge((X, Y), complete_explanation), not _xclingo_visible(X).
_xclingo_skip(X, Y) :- _xclingo_edge((X, Y), complete_explanation), not _xclingo_visible(Y).
```

```
%
_xclingo_reach(X, Z) :- _xclingo_skip(X, Z).
_xclingo_reach(X, Z) :- _xclingo_reach(X, Y), _xclingo_skip(Y, Z), not _xclingo_visible(Y).
%

% Explanation (compressed) graph
_xclingo_graph(explanation).
_xclingo_edge((Caused, Cause), explanation) :- _xclingo_edge((Caused, Cause), complete_explanation), not _xclingo_skip(
      Caused, Cause).
_xclingo_edge((Caused, Cause), explanation) :- _xclingo_reach(Caused, Cause), _xclingo_visible(Caused), _xclingo_visible(
      Cause).
_xclingo_edge((ToExplainAtom, Cause), explanation) :- _xclingo_reach(ToExplainAtom, Cause), _xclingo_visible(Cause),
      _xclingo_show_trace(ToExplainAtom).
_xclingo_node(Caused, explanation) :- _xclingo_visible(Caused).
_xclingo_node(ToExplainAtom, explanation) :- _xclingo_show_trace(ToExplainAtom).

% Labels
_xclingo_attr(node, Atom, label, Label) :- _xclingo_label(Atom, Label), _xclingo_node(Atom, explanation).
```

Listing B.2: Complete contents of logic program *xclingo_graph.lp*.

```
%%%%%%%%%%%%%% xclingo_show.lp %%%%%%%%%%%%%%%%%
#show.
% All show traces
#show _xclingo_show_trace(Atom) : _xclingo_show_trace(Atom), _xclingo_node(Atom, Graph), Graph=complete_explanation.
#project _xclingo_show_trace(Atom) : _xclingo_show_trace(Atom), _xclingo_node(Atom, Graph), Graph=explanation.
% Which causes explain not visible show_traces
% #show _xclingo_link(ToExplainAtom, Cause) : _xclingo_link(ToExplainAtom, Cause).
% #project _xclingo_link(ToExplainAtom, Cause) : _xclingo_link(ToExplainAtom, Cause).

% Edges of the explanation
#show _xclingo_edge((Caused, Cause), Graph) : _xclingo_edge((Caused, Cause), Graph), Graph=explanation.
#project _xclingo_edge((Caused, Cause), Graph) : _xclingo_edge((Caused, Cause), Graph), Graph=explanation.
% Labels
#show _xclingo_attr(Type, Atom, Attr, Label) : _xclingo_attr(Type, Atom, Attr, Label), Type=node, Attr=label, _xclingo_node(
      Atom, explanation).
#project _xclingo_attr(Type, Atom, Attr, Label) : _xclingo_attr(Type, Atom, Attr, Label), Type=node, Attr=label,
      _xclingo_node(Atom, explanation).
```

Listing B.3: Complete contents of logic program *xclingo_show.lp*.

Appendix C

Additional Examples

C.1 An example on explaining constraints

Consider the following example in Figure C.1. In this example, an agent has to move from an initial position to a goal position using an exact number of moves. For the particular number of 4 steps, the

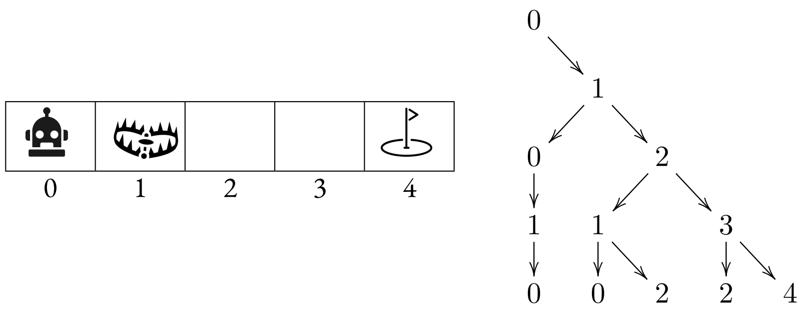


Figure C.1: Description of an instance of the problem of moving an agent towards a goal. On the left side, we see the scenario for the particular instance. On the right side, we see the complete tree of possible movements when the number of admitted steps is 4.

possible paths that the agent can make starting from the initial position 0 are a total of 5 different paths. All of them are depicted on the right side of Figure C.1. Note that only one of them ends at the goal position 4. Additionally, the scenario contains a trap at position 2 so the agent must avoid it when planning its route to the goal. It is easy to see how for this example there is no route such that this latter premise is not broken.

The Program P C.1 implements the example. Lines from 3 to 10 specify the domain and the problem instance. In particular, lines 8, 9, and 10 represent the position of the trap, the goal position and the initial position of the agent, respectively. A choice rule (in lines 12 to 16) is used to generate the different movement plans. The generated movements are *traced* using the annotations in lines 18 and 19. Finally, line 25 forbids the agent to cross any cell with a trap on it at any step, while line 28 forbids any movement plan that does not end in the goal position. Note how these two constraints are *traced* by a `trace_rule` annotation.

```
% trapped_agent.lp

%% Domain and instance
#const time=4.
timestep(0..time).
cell(0..4).

trap_at(1).
goal_at(4).
pos(0, 0).

%% Movement
1{
    pos(X', T): X'=X+1, cell(X'), pos(X, T-1);
    pos(X', T): X'=X-1, cell(X'), pos(X, T-1)
}1 :- timestep(T), T>0.

%!trace {pos(Pos, T), "Step %: agent moves to %", T, Pos} :- pos(Pos, T).
%!trace {pos(Pos, T), "Trapped!"} :- pos(Pos, T), trap_at(Pos).


%% Constraints

%!trace_rule {"Oh no! Trapped", T, TrapPos}.
:- pos(TrapPos, T), trap_at(TrapPos), timestep(T).

%!trace_rule {"In final step % Agent is not at goal %", time, GoalPos}.
:- goal_at(GoalPos), not pos(GoalPos, time).
```

Program C.1: Annotated ASP code implementing the example from Figure C.1.

Recalling the possible paths the agent can choose from the right side of Figure C.1, it is easy to see how all of them fail to comply with the trap constraint, while only one of them ends in position 4 meeting the goal condition. Therefore, this instance is UNSATISFIABLE, and solving it would not produce any answer set. However, since both constraints are *traced*, once `xclingo` finds out the original program is UNSAT, it will relax those and try to solve it again, obtaining 5 answer sets, each one of them featuring one of the 5 possible paths. Instead of trying to explain why the constraints are being violated in all the answer sets, `xclingo` tries to find the one with the smallest number of them. This minimization leads to a sequence of explained answer sets until we reach an optimum. This last answer set, informally speaking, would be the closest one to be a valid answer to the problem. The particular output for Program P C.1 is shown in Listing C.1.

```
UNSATISFIABLE
Relaxing constraints... (mode=minimize)
Answer: 1
##Explanation: 1.1
    *
    |__"Oh no! Trapped"
    |  |__"Step 1: agent moves to 1";"Trapped!"
    |  |  |__"Step 0: agent moves to 0"

    *
    |__"Oh no! Trapped"
    |  |__"Step 3: agent moves to 1";"Trapped!"
    |  |  |__"Step 2: agent moves to 0"
    |  |  |  |__"Step 1: agent moves to 1";"Trapped!"
    |  |  |  |  |__"Step 0: agent moves to 0"

    *
    |__"In final step 4 Agent is not at goal 4"

##Total Explanations:   1
Answer: 2
##Explanation: 2.1
    *
    |__"Oh no! Trapped"
    |  |__"Step 1: agent moves to 1";"Trapped!"
    |  |  |__"Step 0: agent moves to 0"

##Total Explanations:   1
Models: 2
```

Listing C.1: Output for Program P C.1 after running Command C.1

The output above was obtained using the following command:

```
xclingo -n 0 0 trapped_agent.lp
```
(Command C.1)

On it, you can see how the first try to solve the program produces UNSAT (line 1). Then `xclingo` relaxes the constraints and solves them again. At first, it explains an answer set where constraints are violated a total of 3 times: the agent crosses a trap 2 times and its path does not end at the goal position. After that, it finds another answer where the trap constraint is only violated 1 time and the goal constraint is not violated. The first answer corresponds to the leftmost branch of the planning tree from Figure C.1 and the latter corresponds to the rightmost path where the agent ends in position 4.

C.2 An example on diagnosis using `xclingo`

We consider an example from [11] (Fig. C.2). In the example, an analog AC circuit is presented. In it, an agent can close a switch that should ultimately cause a bulb to turn on. However, there are exogenous actions that can modify the environment and make the bulb not turn on by closing the switch. Our goal is to develop a diagnostic system that can identify the reasons why the light does not turn on and present them to the user in the form of readable explanations.

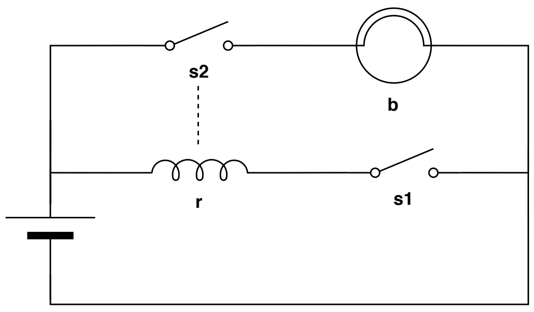


Figure C.2: A circuit with a bulb b, a relay r and two switches, $s1$ and $s2$.

Example 6. (From [11]) *Consider a system S consisting of an agent operating an analog circuit AC from Fig. C.2. We assume that switches s_1 and s_2 are mechanical components that cannot become damaged. Relay r is a magnetic coil. If not damaged, it is activated when s_1 is closed, causing s_2 to close. Undamaged bulb b emits light if s_2 is closed. For simplicity of presentation, we consider the agent capable of performing only one action, $close(s_1)$. The environment can be represented by two damaging exogenous[1] actions: brk, which causes b to become faulty, and srg (power surge), which damages r and also b assuming that b is not protected. Suppose that the agent operating this device is given the goal of lighting the bulb. He realizes that this can be achieved by closing the first switch, performing the operation, and discovering that the bulb is not lit.* □

Our ASP implementation of this example (we call program P_1 in Listings C.2 and C.3) follows the one presented in [11] with the addition of `c/3` and few other predicates for improving the explanation results. At first sight, the encoding may seem too involved for our small example, but this is because the representation is general enough to cover a whole family of similar diagnosis problems. Listing C.2 contains the basic type definitions. The predicate names are self-explanatory, except perhaps lines 15–24. This is because we allow arbitrary fluent domains that can be specified explicitly through predicate `value(F,V)`, meaning that fluent `F` may have value `V`. When no value has been specified in that way, fluents are assumed Boolean by default. Finally, predicate `domain(F,V)` collects all domain values for `V`, regardless of whether they are defined explicitly or by default. In our example, fluents `relay`, `light`, `s1` and `s2` can take values `on` and `off` (to make them more readable) whereas the rest of fluents are Boolean.

Listing C.3 contains the description of the problem. Given any action `A`, fluent `F`, value `V` and time point `I` we use the following predicates:

`h(F,V,I)`	= `F` holds value `V` at `I`
`obs_h(F,V,I)`	= `F` was observed to hold value `V` at `I`
`c(F,V,I)`	= `F`'s value was caused to be `V` at `I`
`c(F,I)`	= `F`'s value was caused at `I`
`o(A,I)`	= `A` occurred at `I`
`obs_o(A,I)`	= `A` was observed to occur at `I`

As usual in diagnosis problems, we differentiate between what happens in the real world, with predicates `h/3` and `o/2`, and the partial observations we have about that world, with predicates `obs_h/3` and `obs_o/2`, respectively. If we execute `clingo` on this code, we obtain the three answer sets that correspond to the possible diagnosis: one including an exogenous action `o(break,1)`; a second one with the exogenous action `o(surge,1)`; and, finally, a third, non-minimal diagnosis where both exogenous actions occur. Of course, in the original work by Balduccini and Gelfond, diagnoses were additionally minimized to avoid the unnecessary addition of exogenous actions, but for the purpose of this paper, we consider the three answer sets of program P_1 as equally interesting for generating explanations. We will complete it using different `xclingo` features in order to get the diagnoses in a fully readable and understandable way.

First, in Listing C.4 we introduce some `trace_rule` annotations for the *indirect effects* and *malfunctioning* rules we introduced in Listing C.3.

```
% plength(1).
plength(1).
time(0..L) :- plength(L).
step(1..L) :- plength(L).

switch(s1). switch(s2).
component(relay). component(bulb).

fluent(relay).
fluent(light).
fluent(b_prot).
fluent(S):-switch(S).
abfluent(ab(C)) :- component(C).
fluent(F) :- abfluent(F).

value(relay,on). value(relay,off).
value(light,on). value(light,off).
value(S,open) :- switch(S).
value(S,closed) :- switch(S).
hasvalue(F) :- value(F,V).
% Fluents are boolean by default
domain(F,true) :- fluent(F), not hasvalue(F).
domain(F,false) :- fluent(F), not hasvalue(F).
% otherwise, they take the specified values
domain(F,V) :- value(F,V).


agent(close(s1)).
exog(break).
exog(surge).
action(Y):-exog(Y).
action(Y):-agent(Y).
```

Listing C.2: Type predicates for program P_1.

```
% Inertia
h(F,V,I) :- h(F,V,I-1), not c(F,I), step(I).

% Axioms for caused
h(F,V,J) :- c(F,V,J).
c(F,J)   :- c(F,V,J).

% Direct effects
c(s1,closed,I) :- o(close(s1),I), step(I).

% % Indirect effects
c(relay,on,J)   :- h(s1,closed,J), h(ab(relay),false,J), time(J).
c(relay,off,J)  :- h(s1,open,J), time(J).
c(relay,off,J)  :- h(ab(relay),true,J), time(J).
c(s2,closed,J)  :- h(relay,on,J), time(J).
c(light,on,J)    :- h(s2,closed,J), h(ab(bulb),false,J), time(J).
c(light,off,J)   :- h(s2,open,J), time(J).
c(light,off,J)   :- h(ab(bulb),true,J), time(J).

% Malfunctioning
c(ab(bulb),true,I) :- o(break,I), step(I).
c(ab(relay),true,I) :- o(surge,I), step(I).
c(ab(bulb),true,I) :- o(surge,I), not h(b_prot,true,I-1), step(I).


% Executability
:- o(close(S),I), h(S,closed,I-1), step(I).

% Something happening actually occurs
o(A,I) :- obs_o(A,I), step(I).

% Check that observations hold
:- obs_h(F,V,J), not h(F,V,J).

% Completing the initial state
h(F,V,0) :- domain(F,V), not -h(F,V,0).
-h(F,V,0) :- h(F,W,0), domain(F,V), W!=V.


% A history
obs_h(s1,open,0).
obs_h(s2,open,0).
obs_h(b_prot,true,0).
obs_h(ab(bulb),false,0).
obs_h(ab(relay),false,0).
obs_o(close(s1),1).

% Something went wrong
obs_h(light,off,1).

% Diagnostic module: generate exogenous actions
o(Z,I) :- step(I), exog(Z), not no(Z,I).
no(Z,I) :- step(I), exog(Z), not o(Z,I).
```

Listing C.3: Program P_1 describing Example 6.

```
%%%%%% Indirect effects
%!trace_rule {"The relay is working at %",J}.
c(relay,on,J)    :- h(s1,closed,J), h(ab(relay),false,J), time(J).

%!trace_rule {"The relay is not working at %",J}.
c(relay,off,J)  :- h(s1,open,J), time(J).

%!trace_rule {"The relay is not working at %",J}.
c(relay,off,J)  :- h(ab(relay),true,J), time(J).

c(s2,closed,J)  :- h(relay,on,J), time(J).

%!trace_rule {"The light is on at %",J}.
c(light,on,J)     :- h(s2,closed,J), h(ab(bulb),false,J), time(J).

%!trace_rule {"The light is off at %",J}.
c(light,off,J)    :- h(s2,open,J), time(J).

%!trace_rule {"The light is off at %",J}.
c(light,off,J)    :- h(ab(bulb),true,J), time(J).

%%%%%% Malfunctioning
%!trace_rule {"The bulb has been damaged at %",I}.
c(ab(bulb),true,I) :- o(break,I), step(I).

%!trace_rule {"The relay has been damaged at %",I}.
c(ab(relay),true,I) :- o(surge,I), step(I).

%!trace_rule {"The bulb has been damaged at %",I}.
c(ab(bulb),true,I) :- o(surge,I), not h(b_prot,true,I-1), step(I).
```

Listing C.4: Adding trace labels to specific rules with `trace_rule`.

This is a good opportunity to emphasize the utility of `trace_rule`. As we can see, we have several rules for the same effects. For instance, we have lines 6 and 9 where we can find rules that capture when the relay becomes off. However, they capture two different cause-effect relations, that are easily understood by just reading their corresponding text traces. Having traced this using `trace` annotations, we would have not been able to differentiate between the activation of both rules, thus we would have not been able to appropriately represent the explanations.

```
%!trace { o(surge,J), "Hypothesis: there has been a power surge at %",J} :-  o(surge,J).
%!trace { o(break,J), "Hypothesis: something has broken the bulb at %",J} :-  o(break,J).
%!trace { o(close(s1),J), "The agent has closed switch s1 at %",J} :-  o(close(s1),J).

%!trace { h(ab(C),true,0), "The % was initially damaged",C} :-  h(ab(C),true,0).
%!trace { h(ab(C),false,0), "Initially, the % was not damaged",C} :-  h(ab(C),false,0).

%!trace {h(F,V,0), "% was initially %",F,V} :- h(F,V,0), not abfluent(F).
```

Listing C.5: Tracing atoms through `trace` annotations for Program P_1.

Additionally, in Listing C.5 we introduce some `trace` annotations for the occurred actions and for the initial values of the fluents. And finally, we request `xclingo` to explain the value of the funds

```
%!show_trace {h(light,V,1)} :- h(light,V,1).
%!show_trace {h(relay,V,1)} :- h(relay,V,1).
```

Listing C.6: Choosing which atoms to explain through `show_trace` annotations for Program P_1.

light and *relay* by requesting the causes for the atoms `h(light, V, 1)` and `h(relay, V, 1)`. This is done through a couple of `show_trace` annotations shown in Listing C.6.

Fig. C.3 shows the explanations for the three answer sets we would obtain with plain `clingo`. Answer set 1 corresponds to the case in which both a power surge occurred and something broke the bulb. The explanation for the relay being off (lines 2–6) can be read as follows: "the relay is not working at 1 *because* it has been damaged *because* there has been a power surge." We have started all explanations for exogenous actions with the word `Hypothesis` to clarify that these are assumptions added to explain the observations. As we can see, in this first answer set, there are two alternative valid causes for the light being off. The first one (Fig. C.3, lines 9–12) is that the bulb was damaged because something broke it. The three lines in the explanation respectively come from the annotations (we will see later) for lines 18, 21 and 52 in Listing C.3. The second cause (lines 14–16) is that the light was already off in the initial state because switch `s2` was initially open: in this case, because of the activation of rules in lines 17 and 5 in Listing C.3.

Answer set 2 (lines 19–29) corresponds to the case in which we just had a power surge. When this happens, the relay is not working (as in the answer set 1) and the light simply remains off, since `s2` was initially open. In this case, we do not get the additional reason for having the light off, since the bulb is not broken.

Finally, answer set 3 shows the case where something breaks the bulb but there is no power surge. In this case, we can see that the relay eventually worked because the agent closed the switch 1 and the relay was not initially damaged. As nothing else happens, the relay is still undamaged in state 1.

```
Answer: 1
##Explanation: 1.1
  *
  |__"The light is off at 1"
  |  |__"The bulb has been damaged at 1"
  |  |  |__"Hypothesis: something has broken the bulb at 1"

  *
  |__"The relay is not working at 1"
  |  |__"The relay has been damaged at 1"
  |  |  |__"Hypothesis: there has been a power surge at 1"

##Explanation: 1.2
  *
  |__"The light is off at 1"
  |  |__"s2 was initially open"

  *
  |__"The relay is not working at 1"
  |  |__"The relay has been damaged at 1"
  |  |  |__"Hypothesis: there has been a power surge at 1"

##Total Explanations:   2
Answer: 2
##Explanation: 2.1
  *
  |__"The light is off at 1"
  |  |__"s2 was initially open"

  *
  |__"The relay is not working at 1"
  |  |__"The relay has been damaged at 1"
  |  |  |__"Hypothesis: there has been a power surge at 1"

##Total Explanations:   1
Answer: 3
##Explanation: 3.1
  *
  |__"The light is off at 1"
  |  |__"The bulb has been damaged at 1"
  |  |  |__"Hypothesis: something has broken the bulb at 1"

  *
  |__"The relay is working at 1"
  |  |__"The agent has closed switch s1 at 1"
  |  |__"Initially, the relay was not damaged"

##Total Explanations:   1
Models: 3
```

Figure C.3: Explanations obtained for the annotated version of P_1.

```
Answer: 1
##Explanation: 1.1
  *
  |__"light is off at 20"
  |  |__"The light is off at 20"
  |  |  |__"s2 is open at 20"
  |  |  |  |__"s2 is open at 19"
  |  |  |  |  |__"s2 is open at 18"
  |  |  |  |  |  |__"s2 is open at 17"
  |  |  |  |  |  |  |__"s2 is open at 16"
  |  |  |  |  |  |  |  |__"s2 is open at 15"
  |  |  |  |  |  |  |  |  |__"s2 is open at 14"
  |  |  |  |  |  |  |  |  |  |__"s2 is open at 13"
  |  |  |  |  |  |  |  |  |  |  |__"s2 is open at 12"
  |  |  |  |  |  |  |  |  |  |  |  |__"s2 is open at 11"
  |  |  |  |  |  |  |  |  |  |  |  |  |__"s2 is open at 10"
  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"s2 is open at 9"
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"s2 is open at 8"
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"s2 is open at 7"
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"s2 is open at 6"
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"s2 is open at 5"
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"s2 is open at 4"
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"s2 is open at 3"
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"s2 is open at 2"
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"s2 is open at 1"
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |__"s2 is open at 0";"s2 was initially open"

##Total Explanations:   1
Models: 1
```

Figure C.4: Explanation for `h(light, off, 20) when only a surge occurred at time step 1 and inertia is traced.`

One important decision in terms of explanation design in the context of diagnosis is to avoid tracing the inertia. In particular, for this example, we have the inertia rule from Listing C.3, Line 2. Actually, having traced this rule would cause the explanations to grow in size. For instance, if we ask for the explanation of `h(light, off, 20)` in answer set 2 of Figure C.3 (replacing time stamp 1 by 20), we still get the same explanation since the switch was initially off and *nothing else changed that* in the whole time interval. However, if we trace inertia, for example by introducing a `trace` annotation like:

```
%!trace {h(F,V,T), "% is % at %",F,V,T} :- h(F,V,T).
```

for showing the value of each fluent at each time step, we would end up producing an explanation like what we can see in Listing C.4 Where the explanation lineally grows in size but without adding any relevant information to the user.

Appendix D

Liver Transplantation Dataset: Description of the Features

Table D.1: Feature Descriptions

Feature Name	Type:units	Description
don_noradrenaline	Numeric:µg (base)/kg/min	Dose of noradrenaline administered to the donor
don_sodium	Numeric:mEq/L	Donor's blood level of sodium
rec_weight	Numeric:kg	Recipient's weight
rec_afp	Numeric:ng/mL	Recipient's blood level of alpha-fetoprotein
rec_inr	Numeric	Standardized measure of Recipient's blood clotting, which is measured in dimensionless units.
don_acv	Boolean	True when the Donor's cause of death was CVA
rec_hypertension	Boolean	True when the recipient is hypertensive
rec_hcv	Boolean	True when the recipient is HCV-positive or HCV-negative.
rec_hcc_afp_30	Boolean	True whenever (1) the recipient is HCC-positive and (2) the Recipient's alpha-fetoprotein is greater than 30 ng/ml. This feature aims to capture the presence of aggressive HCC in the Recipient's body
rec_provenance	Categorical	The situation of the recipient prior to the transplantation procedure. It has the following possible values: *Home*, *Admitted in Ward* or *Admitted in ICU*

Appendix E

Extended Summary in Spanish

La Inteligencia Artificial (IA) ha avanzado notablemente en los últimos años, logrando realizar tareas que antes se pensaba que eran exclusivas de los humanos. La IA generativa [23], por ejemplo, puede producir texto en lenguaje natural, imitar voces y generar diversos tipos de contenido visual, incluyendo imágenes realistas y modelos 3D. Este rápido progreso ha generado una búsqueda de aplicaciones positivas en áreas como la seguridad en redes sociales, la educación personalizada y la traducción en tiempo real. Sin embargo, el lado negativo de este avance tecnológico es el potencial para el uso indebido, que incluye manipulación política, cibercrimen y robo de identidad.

Para abordar estas preocupaciones, se están llevando a cabo esfuerzos regulatorios [28], ejemplificados por la adopción por parte de la Unión Europea (UE) de la primera ley integral sobre IA [27] (la llamada *Ley de IA de la UE*). Esta legislación categoriza los sistemas de IA por nivel de riesgo y describe los marcos regulatorios correspondientes. Consideraciones clave incluyen la privacidad de los datos, la equidad y los dominios específicos en los que operan los sistemas de IA, como los sectores legal, educativo y de salud. En particular, de acuerdo con los puntos 1 y 2 del Artículo 13 de la propuesta del Parlamento Europeo, se requiere transparencia y comprensibilidad de los sistemas.

> *Artículo 13.1. Los sistemas de IA de alto riesgo deben diseñarse y desarrollarse de manera que su funcionamiento sea suficientemente transparente para permitir a los usuarios interpretar la salida del sistema y utilizarla adecuadamente. ...*
>
> *Artículo 13.2. Los sistemas de IA de alto riesgo deben ir acompañados de instrucciones de uso en un formato digital apropiado u otro que incluya información concisa, completa, correcta y clara que sea relevante, accesible y comprensible para los usuarios.*

Propuesta de la Comisión Europea sobre la regulación de sistemas de IA [27]. Propuesta en 2021, aprobada en diciembre de 2023.

La investigación en XAI tiene como objetivo hacer que los algoritmos de IA sean transparentes y comprensibles, aprovechando los conocimientos tanto de la informática como de las ciencias sociales. Además, las ciencias filosóficas y éticas resaltan la importancia de esto. Por ejemplo, V. Dignum [34] describe los requisitos para la interpretabilidad de la IA, enfatizando la transparencia, la responsabilidad y la rendición de cuentas (en inglés, *accountability*).

1. Transparencia

   > *Indica la capacidad de describir, inspeccionar y reproducir los mecanismos mediante los cuales los sistemas de IA toman decisiones ...*

2. Responsabilidad

   > *Se refiere al papel de las personas mismas en su relación con los sistemas de IA. ... La responsabilidad en la IA también es un problema de regulación y legislación, en particular en lo que respecta a la responsabilidad legal ...*

3. *Rendición de cuentas*

   > *La rendición de cuentas es la capacidad de dar cuenta, es decir, de poder informar y explicar las acciones y decisiones de uno. Para garantizar la rendición de cuentas, las decisiones deben derivarse de, y explicarse mediante, los mecanismos de toma de decisiones utilizados. ...*
   >
   > *Al desarrollar mecanismos de explicación, es importante tener en cuenta que las explicaciones deben ser comprensibles y útiles para un ser humano, ...*
   >
   > *La explicación es relevante para confiar en los sistemas de IA por varias razones. En primer lugar, las explicaciones pueden reducir la opacidad de un sistema y apoyar la comprensión de su comportamiento y sus limitaciones. ...*

Lamentablemente, tal y como Miller dice en su artículo [75], *"se podría decir que la mayoría del trabajo en inteligencia artificial explicable emplea únicamente la intuición de los investigadores en qué es una buena explicación"*, en lugar de apoyarse en la vasta investigación ya existente en la materia desde las ciencias sociales. Su trabajo se puede resumir en que, para conseguir la confianza de los usuarios, las explicaciones deben ser:

> (1) *Las explicaciones son contrastivas:* la investigación sugiere que las personas tienden a pedir explicaciones en respuesta a observaciones anormales o inesperadas con respecto a sus propias creencias.

En otras palabras, las personas no simplemente preguntan por qué ocurre un evento P. Típicamente, cuando una persona pregunta "¿Por qué P?", esta posiblemente esperaba que P no hubiese ocurrido (P es inesperado para ella) y, en su lugar, esperaban que hubiese ocurrido un evento distinto Q Por lo tanto, la verdadera pregunta a la que necesitan una respuesta es "¿Por qué P en lugar de Q?". Tenga en cuenta que esto es diferente que simplemente preguntar "¿Por qué no Q?", porque la anterior implica que la persona se imaginaba un mundo predeterminado o esperado, que ha sido quebrantado. De hecho, es a partir de las diferencias entre el mundo real (donde P se cumple) y el mundo predeterminado esperado (donde se cumple Q) que las personas comienzan a construir explicaciones. Tome, por ejemplo, el siguiente ejemplo de [75].

> " ... por qué el transbordador Challenger explotó en 1986 (en lugar de no explotar, o tal vez por qué la mayoría de los otros transbordadores no explotan). La explicación de que explotó "por culpa de sellos defectuosos" parece ser una mejor explicación que "había oxígeno en la atmósfera"."

En el ejemplo, P es la explosión del transbordador Challenger, que ocurrió en el mundo real, mientras que Q podría ser que el transbordador no explote, lo cual era algo esperado por el usuario. "Había oxígeno en la atmósfera" no se considera una buena explicación porque es algo cierto tanto el mundo real como el mundo predeterminado asumido por del actor. Por el contrario, el hecho de que los sellos fueran defectuosos se considera anormal con respecto a las suposiciones del actor y, por lo tanto, se percibe como una explicación satisfactoria.

Por supuesto, estrictamente hablando, el oxígeno en la atmósfera es una causa necesaria para que ocurra la explosión, al igual que otras causas no mencionadas (quizás imposibles de contar). De hecho, manejar la cadena completa de causas rápidamente se vuelve imposible para los humanos en escenarios reales. Esto empeora si también intentamos tener en cuenta todos los posibles inhibidores causales que podrían evitar que ocurra la explosión, incluidos los menos relacionados con el mundo real. Por ejemplo, argumentar que no había tormenta el día del lanzamiento del transbordador como una razón para que ocurra la explosión porque no habría despegado en ese caso. Esto se relaciona con las otras dos propiedades identificadas por [75].

> (2) *Las explicaciones son seleccionadas (de manera sesgada)*: las explicaciones rara vez consisten en las causas reales y completas de un evento, sino más bien en una (sesgada) selección de un subconjunto de causas basada en el contexto y varios otros criterios como la proximidad temporal, la necesidad, la suficiencia, la anormalidad, etc.
>
> (3) *Las explicaciones son sociales*: cuando un sistema explica un resultado, ocurre una transferencia de conocimiento del sistema al usuario como parte de una interacción (similar a las personas). Como tal, la información presentada es relativa a las creencias del sistema sobre las creencias del explicador.

En otras palabras, las explicaciones son tanto dependientes del contexto como del usuario. Cuando las personas intentan encontrar las razones de un evento inesperado, primero intentan imaginar escenarios contrastivos "cercanos" que lo expliquen con éxito. Esto se relaciona con el *razonamiento abductivo* y / o la *simulación causal*. Sin embargo, en este proceso, la exploración de los mundos hipotéticos no es arbitraria, sino cuidadosamente guiada por las creencias de la persona sobre la relevancia de las diferentes posibles causas, que cambian según el contexto. En otras palabras, los escenarios contrastivos imaginados no solo son "cercanos" sino "relevantemente cercanos". Esta selección de causas relevantes no solo influye en la búsqueda de mundos contrastivos válidos sino también en la información incluida en última instancia en la explicación, a menudo también adaptada al conocimiento y la perspectiva del receptor. La siguiente cita de [62] (también recuperada de [75]) ilustra esto perfectamente.

> "Hay tantas causas de x como explicaciones de x. Considere cómo la causa de la muerte podría haber sido presentada por el médico como 'hemorragia múltiple', por el abogado como 'negligencia por parte del conductor', por el constructor del carruaje como 'un defecto en la construcción del freno', por un planificador cívico como 'la presencia de arbustos altos en ese giro'. Ninguna es más verdadera que las demás, pero el contexto particular de la pregunta hace que algunas explicaciones sean más relevantes que otras."

Patrones de Descubrimiento Norwood Russell Hanson. Publicado en 1958.

En [75] varios criterios como la proximidad temporal, la necesidad o la suficiencia (entre otros) se identifican como buenos filtros para causas relevantes, mientras que otros como la probabilidad se muestran como menos útiles. De hecho, en su hallazgo principal final, Miller contrasta las probabilidades con la relación causa-efecto al evaluar una buena explicación.

> (4) *Las explicaciones son causales*: aunque la verdad y la probabilidad son importantes en la explicación y las probabilidades realmente importan, referirse a probabilidades o relaciones estadísticas en la explicación no es tan efectivo como referirse a causas. La explicación más probable no siempre es la mejor explicación para una persona, e importante, usar generalizaciones estadísticas para explicar por qué ocurren los eventos es insatisfactorio a menos que venga acompañado de una explicación causal subyacente para la generalización misma.

Sin embargo, lograr sistemas de IA que cumplan estas propiedades es un gran desafío. Si bien se hacen esfuerzos para desarrollar técnicas de explicación, como las SHAP [68] y LIME [88], estos métodos a menudo se basan en correlaciones y no en conocimiento causal. Esta brecha entre el lenguaje estadístico de la IA y la comprensión causal humana contribuye a la falta de confianza en las explicaciones generadas por la IA. Incluso con la aparición de los llamados *Modelos Masivos de Lenguaje* (LLM, por sus siglas en inglés) [25], como GPT, que han demostrado increíbles dotes en la ejecución de tareas que requieren de la compresnión del lenguaje natural, las explicaciones generadas por estos modelos puden resultar muy convincentes pero falaces, socavando la confianza de los usuarios. Algunos investigadores abogan por sistemas de IA auditables y verificables para abordar estos problemas, enfatizando la importancia del conocimiento causal para respaldar las explicaciones.

En este contexto, el campo de la Representación del Conocimiento y Razonamiento Automático (por sus siglas en inglés, KRR) desempeña un papel crucial, proporcionando una base para representar el conocimiento causal en los sistemas de IA. De hecho, el *razonamiento de sentido común*, un aspecto fundamental de KRR, busca enfatizar que debemos programar las máquinas de la forma más cercana posible a cómo pensamos los humanos. Inspirándonos en esto, proponemos la noción de "Explicación de sentido común", basada en el conocimiento causal real y adaptada a la comprensión humana. Proporcionamos una definición abstracta de esta noción donde intentamos capturar los diferentes aspectos enfatizados por las ciencias sociales y filosóficas para conseguir explicaciones que generen confianza en los humanos.

El objetivo principal de esta tesis es desarrollar el concepto de explicación de sentido común dentro del paradigma de KRR. Este esfuerzo se abordará desde un punto de vista práctico, involucrando la creación de herramientas para calcular explicaciones y su aplicación a diversos problemas y sistemas de IA. Específicamente, el enfoque se centrará en el Razonamiento No-Monótono (por sus siglas en inglés, NMR) [74], un campo dentro de la inteligencia artificial que trata el razonamiento bajo incertidumbre e información incompleta, en contraste con la lógica clásica.

La Programación de Conjuntos de Respuestas (ASP, por sus siglas en inglés)[18, 54] emerge como un paradigma prominente para KRR práctico y resolución declarativa de problemas, fundamentado en principios de NMR. ASP proporciona un marco flexible para representar conocimiento a través de programas lógicos, facilitando el cálculo de soluciones basadas en la semántica de modelos estables [57]. Las herramientas basadas en ASP han sido aplicadas en diversos dominios debido a su accesibilidad y eficiencia [44]. El uso de ASP para explicabilidad [48] ha generado un gran interés en los últimos años, especialmente en el contexto de la IA Explicable (XAI). Se han propuesto varios enfoques para explicar programas ASP, incluyendo semántica causal y explicaciones basadas en grafos. Sin embargo, obtener explicaciones que cumplan con los criterios de explicaciones de sentido común presenta desafíos, lo que hace necesario un mayor desarrollo en el campo.

La parte I de esta tesis proporciona e implementa el los formalismos necesarios para obtener explicaciones de sentido común para (y basadas en) programas ASP, además de explorar su aplicación a diversos contextos del campo de representación de conocimiento y explorar un conjunto relevante de las de las aproximaciones similares existentes.

Esta parte comienza con el Capítulo 3, en el cuál se describe una caracterización formal de las explicaciones en términos de grafos construidos en a partir del programa ASP. Bajo este marco, los modelos pueden ser *justificados*, lo que significa que tienen uno o más *grafos de soporte* (en inglés, *Support Graphs*), o *no justificados* en caso contrario. Demostramos que todos los modelos estables son justificados, mientras que, en general, lo contrario no es cierto, al menos para programas disyuntivos. También caracterizamos un par de operaciones básicas en grafos, que llamamos *poda de aristas* (en inglés *edge prunning*) y *olvido de nodos* (en inglés, *node forgetting*), que permiten realizar filtrado de información en las explicaciones. Se da también una especificación en ASP, para generar los grafos de soporte de un conjunto de respuestas dado de algún otro programa ASP, para la cual demostramos su corrección y validez con respecto a las definiciones dadas.

Posteriormente, el Capítulo 4 presenta `xclingo`, una herramienta que implementa dicha especificación, y que permite el cáclulo de los grafos se soporte para programas ASP. La herramienta además extiende el lenguaje ASP con algunas anotaciones llamadas así que ayudan al usuario a diseñar las explicaciones producidas. Se proporcionan una extensa variedad de ejemplos de cómo usar estas anotaciones en la Sección 4.2. Además, `xclingo` se basa en un método de meta-programación o reificación para calcular los grafos de soporte de un programa. Es decir, la semántica de los grafos de soporte se especifica en el programa ASP de `xclingo` que, junto con una reificación del programa original, cuyos conjuntos de respuesta corresponden a los grafos de soporte de un programa. Tanto la especificación de semántica de la reificación como la de los grafos de soporte se explican y discuten en la Sección 4.3. La arquitectura interna de la herramienta y algunas decisiones de diseño de software se discuten en la Sección 4.4. Esto también incluye características avanzadas adicionales como la opción de agregar *extensiones* a `xclingo`. Las extensiones son fragmentos de código ASP que uno puede inyectar en el programa de cálculo de grafos de soporte de `xclingo` de manera que su comportamiento pueda ser extendido de muchas maneras.

Una vez introducida la herramienta, en el Capítulo 5 brindamos nociones sobre cómo utilizala para obtener explicaciones de sentido común. Comenzamos contrastando las diferencias entre explicaciones técnicas y de sentido común, y discutimos además su importancia y el papel que pueden desempeñar para hacer que los sistemas ASP sean tanto transparentes como comprensibles. Después, en la Sección 5.2, hacemos una observación importante sobre las explicaciones que podemos obtener a partir de programas fuertemente equivalentes. En particular, dos programas ASP fuertemente equivalentes pueden producir explicaciones diferentes. Esto tiene consecuencias importantes que se discuten a lo largo del capítulo, como que las explicaciones de codificaciones ASP eficientes pueden no ser adecuadas para explicaciones de sentido común incluso si obtienen las mismas soluciones que las codificaciones originales. En línea con demostrar esto, proporcionamos un ejemplo práctico en la Sección 5.3 y también una solución práctica al problema en la Sección 5.4.

Tal y como ya se ha introducido, la mayoría de las explicaciones solicitadas por los humanos no son positivas, *"¿Cómo es que p es verdadero?"* sino más bien consultas causales más complejas como preguntas contrastivas que requieren respuestas contrafácticas. La Sección **??** discute más a fondo estos temas y propone el diseño de un sistema que pueda responder a este tipo de consulta causal, incluidas las respuestas a preguntas de tipo *¿Por qué no?*. Se proporciona un ejemplo práctico de cómo implementar este sistema utilizando `xclingo`.

Para finalizar la Parte III, en el Capítulo 6, revisamos algunos de los enfoques más relevantes de la literatura sobre explicaciones en ASP, incluyendo formalismos y herramientas. Realizamos una comparación entre cada enfoque y el nuestro bajo tres perspectivas. Primero, comparamos los objetos matemáticos que respaldan las diferentes definiciones de explicación. Segundo, para aquellos enfoques que proporcionan herramientas utilizables, comparamos sus funcionalidades entre sí y con `xclingo`. Tercero, siempre que sea posible, comparamos los enfoques en términos de la posibilidad de obtener explicaciones de sentido común.

En contraste, la Parte II se centra, en lugar de obtener explicaciones para ASP, en obtener explicaciones de sentido común para algoritmos de aprendizaje automático (por sus siglas en inglés, ML). Esta parte consta de dos capítulos, cada uno dedicado a un problema real y en un dominio distinto, en el que se requiere una solución de explicabilidad para modelos de ML dirigidos a usuarios reales no técnicos.

En concreto, el Capítulo 7 documenta el desarrollo de un Sistema de Apoyo a la Decisión para la combinación de donante-receptor de hígado en trasplantes de hígado. En un contexto crítico como este, no hay discusión sobre por qué se necesitan explicaciones. Además, es particularmente importante que dichas explicaciones sean reproducibles y que proporcionen responsabilidad y transparencia. El sistema desarrollado proporciona estimaciones de la supervivencia de posibles pares donante-receptor, así como explica dicha estimación. Las explicaciones se obtienen utilizando `xclingo` a partir de una representación ASP de un clasificador de ML, en particular un Clasificador de Árbol de Decisión (DT). Más precisamente, se desarrolló una herramienta llamada `Crystal-tree`, que actúa como cliente de la API de Python de `xclingo`, que proporciona explicaciones en lenguaje natural de modelos DT entrenados. En este capítulo, presentamos dicha herramienta y explicamos cómo funciona, incluyendo la implementación basada en ASP utilizando `xclingo`.

En contraste con el capítulo anterior, en el Capítulo 8, nos enfrentamos a la tarea de obtener explicaciones para clasificadores no simbólicos. En concreto, estudiamos la aplicación de un método para obtener explicaciones simbólicas para cualquier tipo de clasificador de ML. Proporcionamos una implementación ASP de dicho método llamado `aspBEEF` y lo probamos para explicar las predicciones de un modelo de apoyo a la invesitigación de desarrollo de medicamentos impresos en 3D. En las próximas secciones, primero presentamos el caso de estudio de investigación de medicamentos impresos en 3D (Sección 8.2) y después presentamos la herramienta aspBEEF y el método en el que se inspira (Sección 8.3).

En conclusión, esta tesis contribuye a la caracterización formal de las explicaciones de los programas lógicos, particularmente definiendo las nociones de gráficos de soporte y modelos justificados. Hemos demostrado la relación entre los modelos estables y los modelos justificados, demostrando que si bien todos los modelos estables están justificados, lo contrario no siempre es cierto, especialmente para programas disyuntivos. Además, hemos introducido operaciones en gráficos, como la poda de aristas y el olvido de nodos, que facilitan el filtrado de información en las explicaciones.

La implementación de la herramienta `xclingo` permite el cálculo de gráficos de soporte para programas ASP. Esta herramienta extiende el lenguaje de Programación de Conjuntos de Respuestas (ASP) con anotaciones para ayudar en el diseño de explicaciones en lenguaje natural. Su implementación en ASP a través de meta-programación asegura la corrección de la herramienta y su mantenibilidad, al tiempo que mejora la usabilidad para los usuarios familiarizados con ASP. Además, la capacidad de la herramienta para calcular solo las explicaciones relevantes mejora considerablemente la eficiencia. Hemos definido explicaciones de sentido común y explicaciones técnicas basadas en la literatura de ciencias sociales, enfatizando la importancia de las explicaciones amigables para el usuario manteniendo el rigor formal. El proceso de diseño de explicaciones, dirigido a obtener explicaciones de sentido común, es una contribución significativa que aborda los desafíos en la obtención de explicaciones adecuadas a partir de programas lógicos equivalentes. Nuestra exploración de preguntas y respuestas causales, junto con la metodología propuesta de pregunta-respuesta, proporciona un marco para procesos de explicación interactivos entre usuarios y sistemas. Esta distinción entre tipos de preguntas y respuestas causales contribuye a clarificar la taxonomía de las explicaciones causales.

Las futuras extensiones de este trabajo incluyen estudiar más a fondo las correspondencias entre gráficos de soporte y otros formalismos, implementar literales causales e incorporar preferencias definidas por el usuario para seleccionar entre explicaciones alternativas. Además, el soporte para causas suficientes y agregados en explicaciones merece una investigación adicional para mejorar las capacidades de la herramienta.